Supplementary Materials of CuckooDuo

1 Algorithm Details

1.1 Pseudocode

Algorithm 1: Insertion workflow of CuckooDuo

```
Input: An item (key-value pair) $e = \langle key, value \rangle$
if $\mathcal{I}_1[h_1(key)]$ or $\mathcal{I}_2[h_2(key)]$ has empty slot then
    Insert $FP(key)$ into the empty slot in CuckooIndex;
    WRITE $e$ into the corresponding slot in CuckooVault;
    return;
$Q$ := An empty queue used for BFS;
for each fingerprint $f$ in $\mathcal{I}_1[h_1(key)]$ and $\mathcal{I}_2[h_2(key)]$ do
    Push $\langle f, path = f \rangle$ into the tail of $Q$;
while $Q$.not_empty do
    Pop $\langle f, path \rangle$ from the front of $Q$;
    if length of path exceeds $L$ then
        return $failure$;
    if $f$ is stored in $\mathcal{I}_1[h_1(f)]$ then
        $h_2(f) = (h_1(f) + hash(f))\%m$;
        if $\mathcal{I}_2[h_2(f)]$ has empty slot then
            Batch READ all items on $path$ from CuckooVault;
            Batch WRITE items on $path$ (and $e$) into CuckooVault;
            Update FPs on $path$ (and $FP(key)$) into CuckooIndex;
            return $success$;
        else if $\mathcal{I}_2[h_2(f)]$ is full then
            for each fingerprint $f'$ in $\mathcal{I}_2[h_2(f)]$ do
                $path' \leftarrow (path, f')$;
                Push $\langle f', path' \rangle$ into the tail of $Q$;
    else if $f$ is stored in $\mathcal{I}_2[h_2(f)]$ then
        $h_1(f) = (h_2(f) - hash(f))\%m$;
        Check $\mathcal{I}_1[h_1(f)]$ and perform similar operations as above;
return $failure$;
```

1.2 Example of Insertion Failure

As shown in Figure 1, we find both candidate buckets of $e_4 = \langle key_4, value_4 \rangle$ are full, and start BFS by enqueuing the four fingerprints in $\mathcal{I}_1[2]$ and $\mathcal{I}_2[3]$. In our example, we set the predefined maximum path length $L = 1$, meaning that our BFS only searches for the kick-out path with the length of one. Next, we sequentially pop the fingerprints from the queue, checking whether each popped fingerprint can be inserted into its alternate candidate bucket. Unfortunately, the alternate candidate buckets of all the four fingerprints are full, and thus we return an insertion failure.

1.3 Details of Multi-threading Acceleration

We describe the details of using multi-threading in conjunction with mutex locking on the slots in CuckooIndex to accelerate the operations of CuckooDuo. We first describe the insertion process. After finding a kick-out path through BFS, we first check each slot on this path to ensure that these slots are not locked. If there are any locked slots on this path, we continue BFS until finding a completely unlocked path. Thereafter, we lock all slots on this path with one atomic operation, followed by executing two batches of RDMA (`READ`

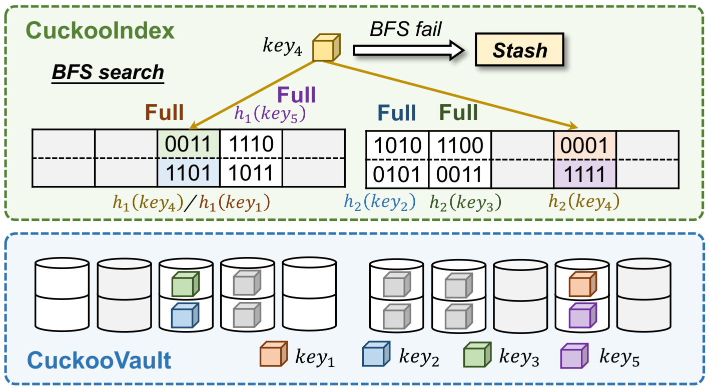
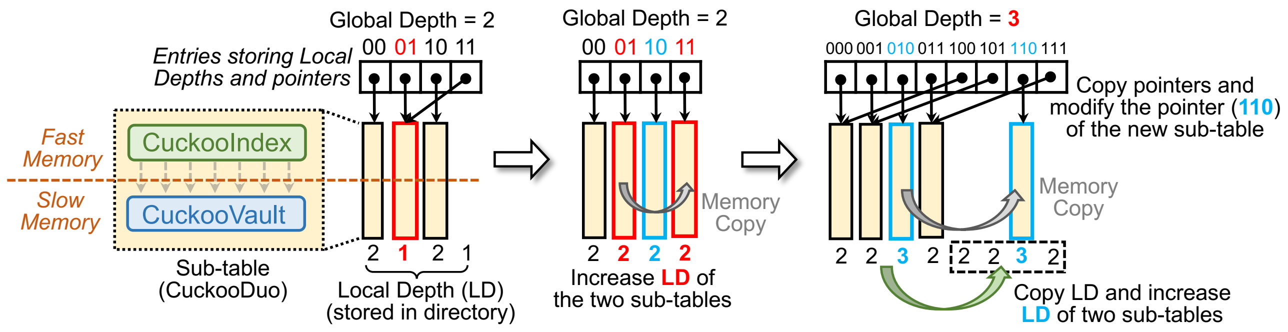


Figure 1: Example of insertion failure ($m = 5$, $d = 2$, $L = 1$).

and WRITE) requests to modify CuckooVault. Once the RDMA WRITE requests are completed, we unlock all slots on the path with an atomic operation. Thanks to the efficiency of BFS, the length of the kick-out path is often very short (<3). In our experiments, setting the maximum kick-out path length $L = 2$ can already attain >95% load factor. Therefore, there will not be many locked slots even under abundant concurrent insertions.

For the hybrid workload with insert/lookup/delete/update operations, we can use multi-threading in conjunction with Read and Write (RW) locks to accelerate CuckooDuo. 1) For lookup, if we find a matched fingerprint in CuckooIndex, we check whether this slot has a Write lock. If so, the lookup operation is blocked until the Write lock is released. Otherwise, the corresponding slot acquires a Read lock, meaning that this slot cannot be modified by insert/update operations until its Read lock is released. 2) For insertion, after finding a kick-out path, we check whether all slots on the path do not have locks. If so, we add Write locks to all slots on the path with an atomic operation. Otherwise, we abandon this path and continue to search for the next shortest path. 3) For deletion, after finding the given key exists in CuckooVault, we check whether its slot in CuckooIndex does not have lock. If so, we delete the fingerprint from CuckooIndex. Otherwise, we wait for its lock to be released and re-execute this operation. 4) For update, after finding the given key exists in CuckooVault, we also check whether its slot in CuckooIndex does not have lock. If so, we add a Write lock to its slot, and send one RDMA WRITE request to update its value. We release the Write lock after the WRITE request is completed.



(a) CuckooDuo with extendible expansion (b) Sub-table expansion (c) Directory expansion

Figure 2: Illustration of CuckooDuo with extendible expansion.

1.4 Details of Extendible Expansion

We explain the extendible expansion of CuckooDuo using the examples used in RACE [1]. As shown in Figure 2, in extendible expansion, the data structure of CuckooDuo includes one directory and multiple sub-tables. The directory is used to index each key into one sub-table. We maintain a global depth GD for the directory, which is initialized as 0. The directory has 2^{GD} entries, each of which points to a sub-table. For example, Figure 2(a)-2(b) show a directory with $GD = 2$ and thus having $2^{GD} = 4$ entries (sub-tables). For each key, we calculate a hash value $h_e(key)$ and use its lower GD bits (*i.e.* $h_e(key)\&((1 << GD) - 1)$) to locate an entry in the directory. Each sub-table (or entry) T_{sub} has a local depth LD, which indicates how many lower bits of $h_e(key)$ are used for indexing the sub-table for each key in T_{sub} (or mapped to this entry). On the other hand, for a sub-table with local depth LD, it is now pointed by 2^{GD-LD} entries in the directory. Besides local depth LD, each entry in the directory also stores a pointer to its sub-table. As shown in Figure 2, in the extendible expansion framework of CuckooDuo, each sub-table is a CuckooDuo consisting of a CuckooIndex (including the stash) and a CuckooVault. The directory along with all CuckooIndexes are stored in fast memory, and all CuckooVaults are stored in slow memory.

When a sub-table is full (*i.e.* an insertion failure occurs in it), we split it into two by adding a new sub-table. As shown in Figure 2(a)-2(b), when the sub-table with suffix "1" is full, we find its local depth $LD = 1$ is not equal to current global depth $GD = 2$. We split this sub-table into two ("01" and "11") by creating a new sub-table. We increase the local depth of entry "01" (old sub-table) from 1 to 2. We also increase the local depth of entry "11" (new sub-table) to 2, and make it point to the new sub-table. The procedure of splitting one sub-table into two is similar as the basic expansion procedure (with expansion ratio of 2) described in Section III-D in our paper, which can also be performed in both active-mode and lazy-mode. Specifically, we first create the new sub-table by copying CuckooIndex and CuckooVault. 1) In active-mode, we sequentially read each bucket in CuckooVault of the old sub-table. For each key, we check the LD-bit suffix of $h_e(key)$ to determine whether it belongs to the old sub-table or the new sub-table, and delete its fingerprint from the CuckooIndex of the sub-table to which it does not belong. 2) In lazy-mode, we mark each bucket in both old sub-table and new sub-table with a 1-bit indicator (stored in local memory). Afterwards, when a marked bucket is first accessed, we incidentally read all items in it from remote memory and clean the redundant items by deleting their fingerprints. As shown in Figure 2(b)-2(c), when the sub-table with suffix "10" is full, we find its local depth $LD = 2$ is equal to global depth $GD = 2$. We increase global depth GD by one, and expand the directory by doubling its size and copying the local depths and the pointers of old entries to the new entries. We also create a new sub-table and split the old sub-table into two as described above. We increase the local depth of entry "010" (old sub-table) and entry "110" (new sub-table) by one, and make entry "110" point to the new sub-table. In this way, except for the entry that the added new sub-table corresponds to (entry "110"), other new entries still point to their corresponding original sub-tables.

By performing extendible expansion, when a sub-table is full, we only need to expand the size of this single sub-table without affecting the items in other sub-tables. In this way, the full-table expansion time is amortized into multiple sub-table expansions. Therefore, extendible expansion avoids excessively long single expansion time, and thus is friendly to latency-sensitive applications. Additionally, extendible expansion also prevents the load factor from halving after each full-table expansion, ensuring consistently high space utilization during the insertion process. Finally, we can see that CuckooDuo can easily perform concurrent lookups during expansion. In our future work, we will design a mechanism that enables CuckooDuo to support concurrent insert, update, and delete operations during expansion.

2 Mathematical Proofs

2.1 Proof for Theorem IV.1

Theorem IV.1. *Consider a basic CuckooDuo under the load factor of α. Let X_α be the number of items failed to be inserted into CuckooDuo due to fingerprint collisions. We have*

$$\mathbb{E}(X_\alpha) \approx 2md^2\alpha^2/2^f \leqslant 4md^2\alpha^2/2^f = O\left(\frac{md^2\alpha^2}{2^f}\right)$$

where m is the number of buckets in each bucket array, d is the bucket size, and f is the length of fingerprints (in bits).

Proof. Consider an incoming item e. The probability that its fingerprint collides with the fingerprint of another item is $1/2^f$. For a CuckooDuo under the load factor of α', the expected number of items in the two candidate buckets of e is $2d\alpha'$. Therefore, the expected probability of item e experiencing a fingerprint collision is

$$\mathcal{P} = 1 - (1 - 1/2^f)^{2d\alpha'} \approx 2d\alpha'/2^f$$

For the CuckooDuo under the load factor of α, we can calculate the expectation of X_α by integrating $\mathcal{P}$ over the number of inserted items x as

$$\mathbb{E}\left(X_\alpha\right) = \int_0^{2dm\alpha} \alpha' \cdot \frac{2d}{2^f} dx = \int_0^{2dm\alpha} \frac{x}{2dm} \cdot \frac{2d}{2^f} dx = \frac{2md^2\alpha^2}{2^f}$$

By fixing the load factor $\alpha' = \alpha$ during integration, we can also derive an upper bound as

$$\mathbb{E}\left(X_\alpha\right) \leqslant 4md^2\alpha^2/2^f$$

□

2.2 Proof for Theorem IV.2

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*

$$\mathbb{E}\left(X\right) \leqslant \frac{4md(d+1)(d-1)}{3 \cdot 2^{2f}} = O\left(\frac{md^3}{2^{2f}}\right)$$

where m is the number of buckets in each bucket array, d is the bucket size, and f is the length of fingerprints (in bits).

Proof. Let d_1 and d_2 be number of slots in each bucket of $\mathcal{I}_1$ using $FP_1(\cdot)$ and $FP_2(\cdot)$ respectively ($d = d_1 + d_2$). We assume $d_1 \geqslant 2$ and $d_2 \geqslant 2$.

Consider a certain bucket $\mathcal{I}_1[i]$ in the first bucket array. If the items in $\mathcal{I}_1[i]$ have only one fingerprint collision (either in FP_1 or FP_2), the collision can definitely be resolved through our *Dual-Fingerprint* adjustment. In the following, we assume that fingerprint collision occurs uniformly across all buckets. This assumption will lead to an upper bound of the failure probability because our *Dual-Fingerprint* algorithm can effectively resolve many collisions.

Let $\mathcal{P}$ be the probability that there exists fingerprint collisions in $\mathcal{I}_1[i]$ that cannot be resolved through *Dual-Fingerprint* adjustment. The upper bound of $\mathcal{P}$ can be written as $\overline{\mathcal{P}} = 1 - \mathcal{P}_0 - \mathcal{P}_1$, where $\mathcal{P}_j$ is the probability that j fingerprint collisions happen in the certain bucket $\mathcal{I}_1[i]$ ($j = 0$ means no fingerprint collision happens).

We derive $\mathcal{P}_0$ and $\mathcal{P}_1$ as

$$\mathcal{P}_0 = \left(\prod_{j=0}^{d-1}\left(1 - \frac{j}{2^f}\right)\right)^2$$

$$\mathcal{P}_1 = 2\left(\prod_{j=0}^{d-1}\left(1 - \frac{j}{2^f}\right)\right) \cdot \binom{d}{2} \cdot \frac{1}{2^f} \cdot \left(\prod_{j=0}^{d-2}\left(1 - \frac{j}{2^f}\right)\right)$$

where $\binom{d}{2} = \frac{d!}{2!(d-2)!}$ is the combination number.

Without loss of generality, we assume that $d < d^2 < d^3 \ll 2^f$. By using mathematical analysis techniques, we can get:

$$\mathcal{P}_0 = 1 - \frac{d(d-1)}{2^f} + \frac{d^2(d-1)^2 - 2/3 \cdot d(d-1)(2d-1)}{2^{2f}} + o\left(\frac{d^3}{2^{2f}}\right)$$

$$\mathcal{P}_1 = \frac{d(d-1)}{2^f} - \frac{d(d-1)^3}{2^{2f}} + o\left(\frac{d^3}{2^{2f}}\right)$$

$$\overline{\mathcal{P}} = 1 - \mathcal{P}_0 - \mathcal{P}_1 = \frac{d(d-1)(d+1)}{3 \cdot 2^{2f}} + o\left(\frac{d^3}{2^{2f}}\right)$$

Recall that we assume $d^3 \ll 2^f$, meaning that $\overline{\mathcal{P}}$ is very small. Therefore, for each item in bucket $\mathcal{I}_1[i]$, the probability that it has a fingerprint collision with another item in $\mathcal{I}_1[i]$ and the collision cannot be resolved with *Dual-Fingerprint* adjustment is $\mathcal{P}/d$.

As each item might collide with all items in its two candidate bucket, the upper bound of the probability that an item experiencing an unresolvable fingerprint collision is $2\overline{\mathcal{P}}/d$. Finally, as there are at most $2dm$ items in CuckooDuo, the expectation of the number of items with unresolvable collisions satisfies that

$$\mathbb{E}(X) \leqslant 2dm \cdot \frac{2\overline{\mathcal{P}}}{d} \leqslant 2dm \cdot \frac{2(d+1)(d-1)}{3 \cdot 2^{2f}} = \frac{4md(d+1)(d-1)}{3 \cdot 2^{2f}}$$

□

2.3 Proof for Theorem IV.3

Theorem IV.3. *Consider a basic CuckooDuo under the load factor of α. Let Y_α 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 m is the number of buckets in each bucket array, d is the bucket size, L is the maximum length of kick-out path, and $\beta(r)$ is the ratio of full buckets under the load factor of r.

Proof. Consider a CuckooDuo under the load factor of r. We define its full bucket ratio as $\beta(r) := \frac{t(r)}{2m}$, where $t(r)$ is the number of full buckets in CuckooDuo. Here, $\beta(r)$ and $t(r)$ are two functions of load factor r.

For an incoming new item e, the probability that it cannot be directly inserted into one of its two candidate buckets is $\mathcal{P}_0 = \beta(r)^2$.

During a BFS process with the maximum kick-out path length of L, there are $2\Sigma_{i=0}^L d^i$ buckets to be checked. We assume the BFS process randomly visits each bucket. The probability that item e cannot be inserted into CuckooDuo through the BFS process is

$$\mathcal{P}_L = \beta(r)^{2\Sigma_{i=0}^L d^i}$$

which is also the probability that the BFS cannot find a non-full bucket.

For the CuckooDuo under the load factor of α, we can calculate the expectation of Y_α by integrating over load factor r.

$$\mathbb{E}\left(Y_\alpha\right) \approx 2md \int_0^\alpha \frac{\mathcal{P}_L}{1-\mathcal{P}_L} dr = 2md \int_0^\alpha \frac{\beta(r)^{2\Sigma_{i=0}^{L} d^i}}{1-\beta(r)^{2\Sigma_{i=0}^{L} d^i}} dr$$

$\square$

3 Additional Experimental Results

3.1 Details about the Testbed

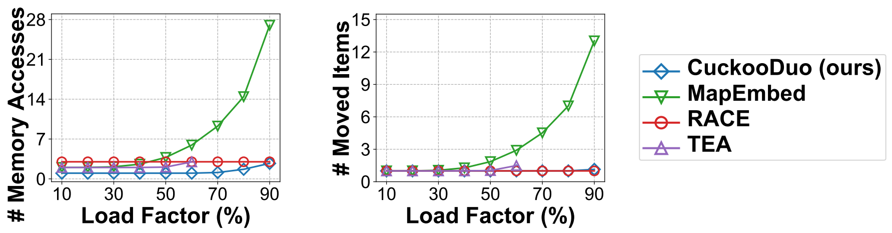
We run all experiments on a testbed with two servers and one switch. Each server is with one 24-core Intel(R) Xeon(R) Silver 4116 @ 2.10GHz CPU, 32GB DRAM, and one 100Gbps Mellanox ConnectX-5 NIC supporting RoCE v2 (RDMA over Converged Ethernet, version 2). Each NIC is connected to a switch via 100Gbps cables. We use one server as the local fast memory, on which we build the hashing index. Another server is used for emulating remote slow memory, on which we build the KV table. To emulate the weak compute power in remote slow memory, we only use the CPU in remote server for registering memory to RNICs during initialization and performing memory copy operation during expansion. We do not use the CPU on remote server for any calculations.

3.2 Comparison of Additional Insertion Properties

We also evaluate the number of memory access and item movement during insertion, and compare the results of four candidate solutions.

Number of memory accesses during insertion (Figure 3(a)): We find that CuckooDuo has the fewest memory accesses during insertion. Under $< 70\%$ load factor, CuckooDuo accesses remote memory only once per insertion. Under 90% load factor, CuckooDuo still only needs 3 memory accesses per insertion, while MapEmbed needs 27 accesses because of frequent data movement.

Data movement during insertion (Figure 3(b)): We find that CuckooDuo achieves minimal data movement across all load factors. We measure the average number of moved items in remote memory (including the incoming item) per insertion under different load factor. The results show that as load factor increases, the data movement of CuckooDuo only slightly increases, while that of MapEmbed significantly increases. Under 90% load factor, CuckooDuo only moves 1.1 items per insertion on average, while MapEmbed needs to move more than 13 items. The number of item movement of RACE is always 1 because it does not move items to improve load factor, and its insertion process only involves writing the incoming item to a destination slot.



(a) # Memory Access per insert (b) # Moved Items per insert

Figure 3: Comparison of insertion properties.

3.3 Cumulative Distribution Function of the Workloads

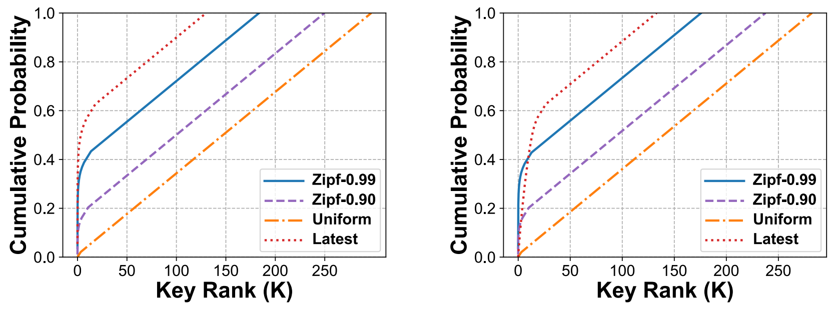
Figure 4 present the Cumulative Distribution Function (CDF) of the lookup/update requests in the workloads used in TABLE III in our paper. We can see that we use the workloads with four different levels of skewness. The distributions used in our paper, ordered by descending skewness, are: latest, zipfian ($\theta = 0.99$), zipfian ($\theta = 0.90$), and uniform.

3.4 Comparison of Performance on Hybrid Workloads

We compare the performance of all candidate works on hybrid workloads with different lookup/insert ratios. We use YCSB to create multiple running workloads with 10M hybrid lookup/insert requests (following default Zipfian distribution with $\theta = 0.99$). We first use the loading workload to load the KV-table to 60% load factor, and then evaluate the performance on these running workloads.

Latency (Figure 5(a)): The results show that the latency of CuckooDuo remains 3.5μs, which is always smaller than that of the other works across different lookup/insert ratios. This is because both of the insert and lookup operation of CuckooDuo can be accomplished in 1 RTT. For the other works, their latency decreases as the increase of lookup/insert ratio, because their lookup latency is smaller than insert latency.

Throughput (Figure 5(b)): We use the multi-threading version of all works described in our paper to evaluate the throughput (under 16 threads). The throughput results are consistent with that of the latency results in Figure 5(a). CuckooDuo always has higher

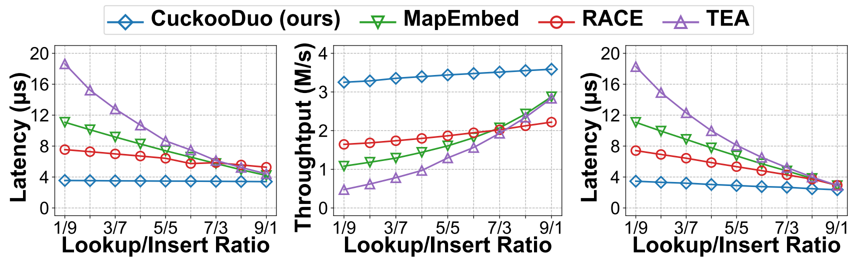


(a) YCSB-A
(Lookup 50%; Update 50%)
(b) YCSB-D
(Lookup 95%; Insert 5%)

Figure 4: Cumulative Distribution Function (CDF) of the workloads used in TABLE III in our paper.

throughput than that of the other works under various lookup/insert ratio. At the lookup/insert ratio of 1/9, the throughput of CuckooDuo, MapEmbed, RACE, and TEA are 3.25, 1.08, 1.64, and 0.47 M/s respectively.

Latency under the same fast memory size (Figure 5(c)): We also evaluate the latency under the setting of same fast memory size to ensure a fair comparison. As described in our paper, we fix the fast memory size of all works to 65MB by adding an 8-way set-associative LRU cache of different sizes to each work. For example, as the index structure (CuckooIndex) of CuckooDuo has the size of 60MB under default setup (using $f = 16$ bit fingerprints), we add a 5MB cache to it. The results have similar trend as the latency results without cache in Figure 5(a). As the cache in fast memory can reduce lookup latency, the latency of all works drops compared to the results in Figure 5(a), especially under large lookup/insert ratios. The results again validate that simultaneously using an index and a cache in fast memory is a promising design.

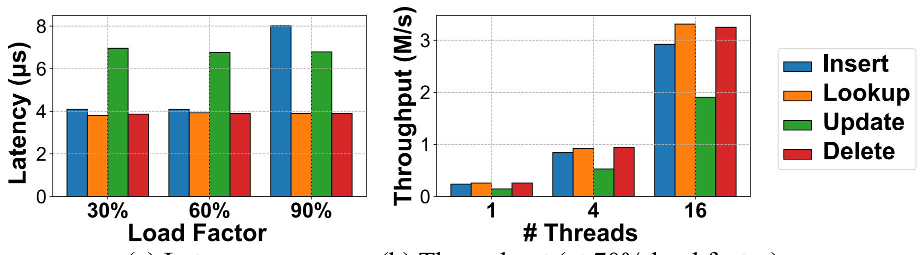


(a) Latency w/o cache (b) Throughput w/o cache (c) Latency w/ cache

Figure 5: Comparison of latency and throughput under hybrid workloads.

3.5 Large-scale Experiments

We conduct large-scale experiments to evaluate the performance of CuckooDuo. We build a CuckooDuo with the KV-table size of 1G. We use YCSB to create a large loading workload with 1G distinct insert requests. We create several running workloads with 10M insert/lookup/update/delete requests. 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. We use the insert workload to load CuckooDuo to different load factor, and then run the four running workloads at different load factors to evaluate the insert/lookup/update/delete latency and throughput. As shown in Figure 6(a), the insert/lookup/update/delete latency is 4.09/3.80/6.96/3.87 μs at 30% load factor, and 8.01/3.90/6.79/3.90 μs at 90% load factor. The latency in large-scale workloads is 0.4$\sim$1.1 μs larger than that in small-scale experiments. Figure 6(b) shows the throughput at 70% load factor. When using 16 threads, the insert/lookup/update/delete throughput is 2.92/3.31/1.90/3.25 M/s, which is 0.1$\sim$0.3 M/s slower than that in small-scale experiments. In large-scale experiments, the speed of CuckooDuo slightly decreases compared to small-scale experiments because the time spent on the local CuckooIndex increases due to more CPU cache misses.



(a) Latency (b) Throughput (at 70% load factor)

Figure 6: Performance on large-scale workloads.

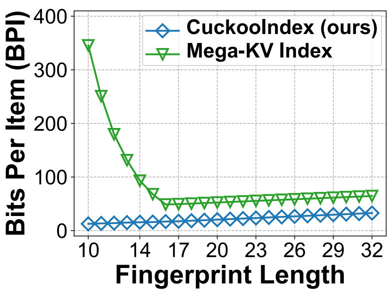
3.6 Space Comparison of CuckooIndex with Indexes Storing Fingerprints and Locations

We compare the space efficiency of CuckooIndex with two indexes storing fingerprints and locations IDs: Mega-KV [2] and Viper [3].

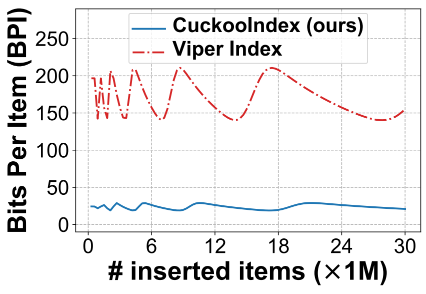
Mega-KV is an excellent in-memory KV store that uses an index implemented on GPU to accelerate the operations. Its index is a cuckoo hash table storing fingerprints of keys and their location IDs. In Figure 7(a), we compare the average index size (bits per item) under different fingerprint length. The results show that as fingerprint length varies from 10 to 16, the BPI of Mega-KV varies from 345 to 50, while the BPI of CuckooIndex varies from 12.5 to 16.9. This is because when using short fingerprints, Mega-KV's index (cuckoo hash table) suffers low load factor incurred by frequent fingerprint collisions. By contrast, our Dual-FP optimization significantly reduces fingerprint collision probability, and thus our CuckooIndex always achieve high load factor and small BPI. When fingerprint length is larger than 16, Mega-KV's index is also about 2× larger than CuckooIndex because it stores item location IDs. By contrast, under our one-to-one mapping design between CuckooIndex and CuckooVault, CuckooIndex does not need to explicitly store the location IDs.

Viper is another efficient KV store designed for DRAM-NVMM architecture. It uses a hash table called CCEH [4] to store the 64-bit fingerprints of keys and their location IDs in persistent memory. CCEH also uses the extendible expansion design for automatically resizing to accommodate larger workloads. Figure 7(b) shows the BPI of Viper's index and our CuckooIndex during the insertion process, where our CuckooIndex also use the extendible expansion mechanism described in § III-D in our paper and we set the sub-table size of both methods to 300. The results show that during the insertion process, the BPI of Viper's index fluctuates from 140 to 210, while the BIP of CuckooIndex fluctuates from 17 to 24, about 8.5× smaller than Viper.

In summary, our CuckooIndex always achieves significantly smaller space overhead than existing indexs storing fingerprints and location IDs. The reason is that 1) our Dual-FP optimization allows using shorter fingerprint length without compromising load factor, and 2) our one-to-one design allows the index to not explicitly store location IDs. We hope our ideas could inspire the design of future fingerprint-based indexes in KV stores.



(a) Comparison with Mega-KV index (cuckoo hash table)



(b) Comparison with Viper index (CCEH)

Figure 7: Comparison with indexes storing fingerprints and locations.

3.7 Experiments under Bandwidth-limited Setting

Setup: As described in § 3.7, our servers use 100Gbps Mellanox ConnectX-5 NIC and 100Gbps cables. The default key/value length is 64-byte. Under this default setting, network bandwidth is sufficient and will not be the bottleneck of latency and throughput. To better highlight the advantages of CuckooDuo, we create the bandwidth-limited setting by reducing the NIC port bandwidth by 10× (from 100Gbps to 10Gbps) and increasing key/value length by 16× (from 64-byte to 1024-byte).

Latency (Figure 8(a)-8(d)): Under this bandwidth-limited setting, CuckooDuo demonstrates a more significant latency improvement. Compared to the other works, CuckooDuo 6.1∼17.6× smaller insert latency, 3.0∼9.8× smaller lookup latency, 3.2∼5.5× smaller update latency, and 6.2∼10.8× delete latency.

Throughput (Figure 8(e)-8(h)): Under this bandwidth-limited setting, CuckooDuo demonstrates a more significant throughput improvement. Compared to the other works, CuckooDuo 8.0∼24.9× higher insert throughput, 8.0∼31.8× higher lookup throughput, 7.6∼30.5× higher update throughput, and 8.0∼31.8× delete latency. It is worth noting that the update throughput of CuckooDuo (and the other works) is only slightly lower than the lookup/delete throughput. However, recall that the update operation of CuckooDuo (and the other works) takes 2 RTTs, and the lookup/delete operation takes 1 RTT. This disparity between the results of RTT and throughput indicates that under such bandwidth-limited setting, throughput is no longer dominated by RTT like in our default setting. Therefore, in such setting, CuckooDuo achieves a more significant throughput improvement because it has smaller bandwidth overhead.

4 Future Directions

We explore some promising future design directions of CuckooDuo.

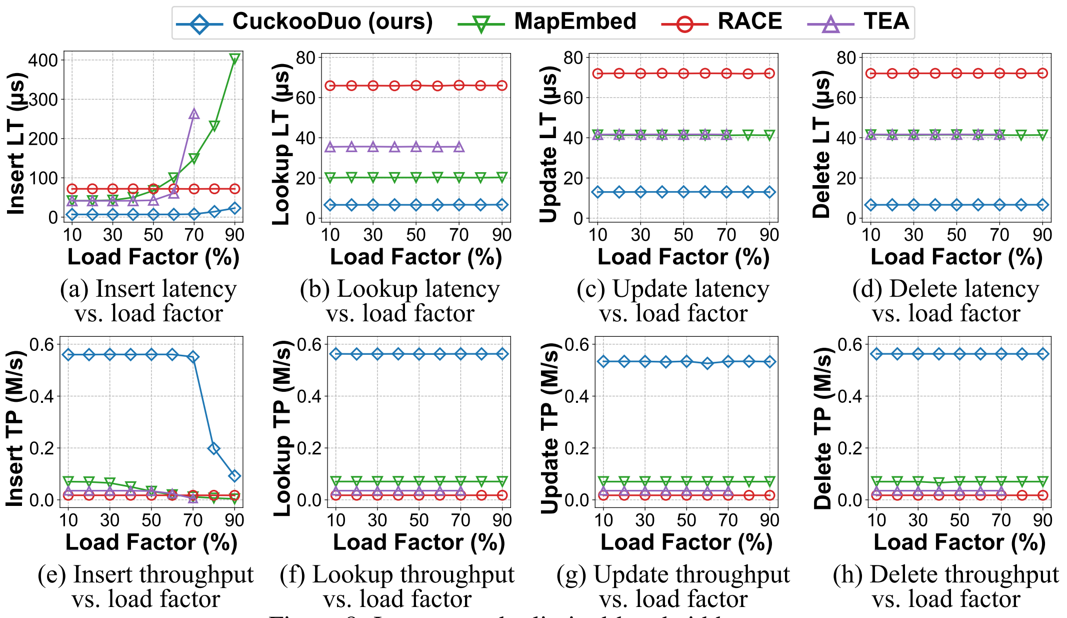


Figure 8: Latency under limited-bandwidth setup.

Variable-sized keys/values: Like many works [5, 6, 7], we only consider fixed-size keys/values in our paper. It is not hard to extend CuckooDuo to support variable-sized keys/values. We can store pointers in CuckooVault to support variable-sized values like prior works [1]. For variable-sized keys, we could also store pointers in CuckooVault. However, as suggested by previous workload analysis in Meta [8] and Twitter [9], small-sized keys dominate in realistic workload. Therefore, we consider using a pointer/key hybrid approach to support variable-sized keys, ensuring both efficient memory usage and flexibility in key management. Specifically, we allow CuckooVault to store small keys directly in buckets, while larger keys be can be referenced via pointers. We add a 1-bit indicator to each fingerprint in CuckooIndex, indicating whether the real short key or the pointer to a long key is stored in the corresponding slot in CuckooVault.

Multiple clients/servers: For now, we only consider the KV-store in fast-slow memory architecture with a single client and server. Extending our solution to the scenario with multiple clients/servers could benefit distributed KV-store servers [10, 11] and more applications in disaggregated memory [1].

We first discuss how to extend CuckooVault to multiple servers. It might seem straightforward to split CuckooVault into multiple parts and deploy each part in each server. However, we should also consider the possibility of uneven request loads among different servers. A potential solution is to use the small cache in the local client to temporarily store requests for hot keys. Under skewed request distribution [12], this approach can significantly reduce the number of requests to remote servers, alleviating the effect of load imbalances and improving overall system performance.

We then discuss how to extend our solution to multi-client scenario. In a typical multi-client non-uniform memory access (NUMA) architecture, delegation [13, 14] and replication [15, 16] are two widely used design strategies. Our CuckooDuo can also adopt these two design approaches. 1) In the delegation design, we store the CuckooIndex in one dedicated server, and let it handle the data requests issued from all clients. However, this centralized delegation design can easily become a bottleneck in large-scale environment. 2) In the replication design, we deploy one CuckooIndex in each client. Existing replication strategies require data coherence protocols to ensure coherent [15, 17, 16]. Of course, we could use existing protocols to actively synchronize CuckooIndexes across all clients with CuckooVault, ensuring perfect coherency. However, we think this approach is overly complex and grossly inefficient. An alternative design is to passively synchronize these CuckooIndexes in a lazy manner. Specifically, the client continues to use its local CuckooIndex (which might be partially incoherent with CuckooVault) to access remote CuckooVault. If an inconsistency is detected (*e.g.*, when an RDMA Compare-And-Swap (CAS) operation fails), the client proactively reads all items from the corresponding remote bucket and synchronize their fingerprints in local CuckooIndex. Unfortunately, in this design, the potentially incoherent CuckooIndex cannot guarantee a worst-case one RTT for lookup operations. However, we believe that for read-intensive workloads, the partially incoherent CuckooIndex can still provide effective guidance for most insertion and lookup operations, significantly improving performance.

Leveraging computation power on remote memory: Inspired by the excellent work called Catfish [18, 19], when remote server has computation power, we could also batch some requests and send them to remote CPU for processing, so as to reduce network load and local CPU pressure. It is trivial to let remote CPU to handle lookup requests. However, if we want to leverage remote CPU to handle insert (update/delete) requests, we also face the consistency challenge that arise from concurrent access to the KV table by both local client requests and the remote backend threads. We let local client to use atomic RDMA operations (Compare-And-Swap (CAS) operations) to modify remote memory, which cannot be interrupted by remote insertion threads. We also let remote threads to use atomic read-and-write operations to modify the KV table. However, the CAS operations of local client might overwrite the keys inserted by remote backend threads. To address this issue, we adopt the re-read approach from RACE [1], letting remote threads to ensure that all insert requests in the batch have been successfully executed. This method guarantees that no inconsistencies are introduced due to concurrent modifications.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Juncheng Yang, Yao Yue, and KV Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)*, 17(3):1–35, 2021.

[10] Redis. `https://redis.io`.

[11] Memcached. `https://memcached.org/`.

[12] 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.

[13] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *Principles of Distributed Systems: 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings 17*, pages 83–97. Springer, 2013.

[14] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364, 2010.

[15] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K Aguilera. Black-box concurrent data structures for numa architectures. *ACM SIGPLAN Notices*, 52(4):207–221, 2017.

[16] Ajit Mathew and Changwoo Min. Hydralist: A scalable in-memory index using asynchronous updates and partial replication. *Proceedings of the VLDB Endowment*, 13(9):1332–1345, 2020.

[17] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-out ccnuma: Exploiting skew with strongly consistent caching. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[18] 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.

[19] 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.