RACE: One-sided RDMA-conscious Extendible Hashing

PENGFEI ZUO, Huawei Cloud, China
QIHUI ZHOU, The Chinese University of Hong Kong, China
JIAZHAO SUN, LIU YANG, and SHUANGWU ZHANG, Huawei Cloud, China
YU HUA, Huazhong University of Science and Technology, China
JAMES CHENG, The Chinese University of Hong Kong, China
RONGFENG HE and HUABING YAN, Huawei, China

Memory disaggregation is a promising technique in datacenters with the benefit of improving resource utilization, failure isolation, and elasticity. Hashing indexes have been widely used to provide fast lookup services in distributed memory systems. However, traditional hashing indexes become inefficient for disaggregated memory, since the computing power in the memory pool is too weak to execute complex index requests. To provide efficient indexing services in disaggregated memory scenarios, this article proposes RACE hashing, a one-sided RDMA-Conscious Extendible hashing index with lock-free remote concurrency control and efficient remote resizing. RACE hashing enables all index operations to be efficiently executed by using only one-sided RDMA verbs without involving any compute resource in the memory pool. To support remote concurrent access with high performance, RACE hashing leverages a lock-free remote concurrency control scheme to enable different clients to concurrently operate the same hashing index in the memory pool in a lock-free manner. To resize the hash table with low overheads, RACE hashing leverages an extendible remote resizing scheme to reduce extra RDMA accesses caused by extendible resizing and allow concurrent request execution during resizing. Extensive experimental results demonstrate that RACE hashing outperforms state-of-the-art distributed in-memory hashing indexes by 1.4–13.7× in YCSB hybrid workloads.

CCS Concepts: • **Information systems** → **Distributed storage**; **Data structures;**

Additional Key Words and Phrases: Disaggregated data center, remote direct memory access, hashing index structure

ACM Reference format:
Pengfei Zuo, Qihui Zhou, Jiazhao Sun, Liu Yang, Shuangwu Zhang, Yu Hua, James Cheng, Rongfeng He, and Huabing Yan. 2022. RACE: One-sided RDMA-conscious Extendible Hashing. *ACM Trans. Storage* 18, 2, Article 11 (April 2022), 29 pages.
https://doi.org/10.1145/3511895

Pengfei Zuo and Qihui Zhou contributed equally to this research.

The preliminary version appears in the Proceedings of the USENIX Annual Technical Conference (USENIX ATC'21) as "One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory".

Authors' addresses: P. Zuo (corresponding author), J. Sun, L. Yang, S. Zhang, and R. He, Huawei Cloud, Shenzhen, Guangdong, China; emails: pfzuo.cs@gmail.com, sunjiazhao@huawei.com, yangliu100@huawei.com, zhangshuangwu@huawei.com, herongfeng@huawei.com; Q. Zhou and J. Cheng, The Chinese University of Hong Kong, Hong Kong, China; emails: qhzhou@cse.cuhk.edu.hk, jcheng@cse.cuhk.edu.hk; Y. Hua, Huazhong University of Science and Technology, Wuhan, Hubei, China; email: csyhua@hust.edu.cn; H. Yan, Huawei, Chengdu, Sichuan, China; email: yanhuabing@huawei.com.

1553-3077/2022/04-ART11 $15.00
https://doi.org/10.1145/3511895

1 INTRODUCTION

Memory disaggregation, which has attracted extensive attentions from both industry, e.g., HP's The Machine [22] and Intel RSD [23], and academia [6, 17, 19, 30, 31, 40], decouples the traditional monolithic compute and memory resources in datacenters and forms independent compute and memory resource pools. Due to resource pooling and independent hardware deployments, disaggregated memory enjoys the benefits of improvements on resource utilization, failure isolation, and elasticity [5, 44]. In the disaggregated memory architecture, compute blades run applications with only a small amount of memory as cache. In contrast, the memory pool stores application data with weak computing power. Due to not involving the compute resources in the memory pool, fast one-sided RDMA networks generally serve for data accesses from the compute blades to the memory pool.

Distributed in-memory hashing indexes have become one of the fundamental building blocks in many datacenter applications, such as databases [25, 29, 47] and key-value stores [2, 3, 27]. With the increasing popularity of RDMA in modern datacenters, **RDMA-search-friendly (RSF)** hashing indexes have been intensively studied, e.g., FaRM hopscotch hashing [14], Pilaf cuckoo hashing [33], and DrTM cluster hashing [46]. These RSF indexes execute search requests by using one-sided RDMA `READ`s to fetch data from remote memory without involving remote CPUs. In contrast, **insertion, deletion, and update (IDU)** requests are sent to the remote CPUs, which locally execute them. However, this mechanism fails to work in the new disaggregated memory architecture, since the computing power in the memory pool is too weak to execute the aforesaid complex IDU requests. In fact, in these RSF hashing indexes, IDU requests can be executed in the compute blades by using one-sided RDMA `WRITE` and `ATOMIC` verbs to operate on remote data. However, we observe that executing IDU requests using one-sided RDMA verbs in existing RSF hashing indexes incurs significant performance degradation, due to a large number of network round-trips and concurrent access conflicts. In a nutshell, it is non-trivial to design an efficient hashing index for disaggregated memory due to the following challenges:

• *Many remote reads&writes for handling hash collisions.* To handle hash collisions, existing hashing schemes incur significant data movement overheads to make room for newly inserted items, e.g., hopscotch hashing [21] and cuckoo hashing [38]. These data movements are executed by many remote reads and writes in the disaggregated memory, which significantly decrease the performance of hashing indexes, since each remote read or write produces one RDMA network round-trip.

• *Concurrency control for remote access.* To handle conflicts of concurrent accesses, lock-based techniques have been widely used in hashing indexes [16, 28]. Locks have low overhead for local hashing indexes, due to nanosecond-level latency for local execution. However, when using locks for hashing indexes in disaggregated memory scenarios, remote locking has to be implemented using RDMA `ATOMIC` verbs with microsecond-level latency, thus incurring high overheads and increasing the waiting delay when lock contention occurs. Especially for the hashing indexes with excessive data movement, multiple locks are acquired before moving data, which exacerbates the lock contention.

• *Tricky remote resizing of hash tables.* When a hash table is full, resizing is inevitable for increasing its size. Conventional full-table resizing needs to move all key-value items from an old hash table to a new one. Extendible resizing [15, 35] reduces the number of moved items during resizing, at the cost of one extra RDMA `READ` due to the need of first accessing the directory of the hash table. Moreover, during resizing, it is challenging to concurrently access the hash table.

To address the above challenges, we propose **RDMA-Conscious Extendible (RACE)** hashing, to the best of our knowledge, the first hashing index designed for disaggregated memory

that fully relies on one-sided RDMA verbs to efficiently execute all index requests. To reduce the performance influence of resizing, RACE hashing leverages the extendible resizing, and hence a RACE hash table consists of multiple subtables and a directory that is used to index subtables. The subtable structure is designed to be one-sided **RDMA-conscious (RAC)**, achieving that all index requests (including search, insertion, deletion, and update) can be executed using only one-sided RDMA verbs while having a constant-scale time complexity in the worst case and therefore delivering high performance. To improve the performance of remote concurrency, RACE hashing leverages a lock-free remote concurrency control scheme for the RAC hash subtable, which achieves that all index requests except failed insertions are concurrently executed in a lock-free manner. Moreover, to reduce the performance penalty from extendible resizing, RACE hashing caches the directory at the client side (a client is a CPU blade) and therefore eliminates the RDMA access to the directory. Nevertheless, since the directory in the client cache becomes stale when the hash table is resized, accessing the hash table via a stale directory cache may obtain incorrect or inconsistent results. RACE hashing presents a simple yet efficient stale-read scheme to guarantee the correctness of accessed data and allow current request execution during resizing.

Specifically, this article makes the following contributions:

• *One-sided RDMA-conscious table structure.* We present a RAC subtable structure that is both RDMA-search-friendly and RDMA-IDU-friendly. All index requests are executed by using only one-sided RDMA verbs with constant worst-case time complexity. IDU requests do not cause any extra data movement.

• *Lock-free remote concurrency control.* We design lock-free remote concurrent algorithms for RACE hashing to enable all requests except failed insertions to be concurrently executed without locking.

• *Extendible remote resizing.* We present a stale-read client directory cache scheme to reduce one extra RDMA `READ` for remote directory lookups and guarantee request execution correctness when using the stale directory cache. We also achieve concurrent access to the subtable that is being resized.

• *Implementation and evaluation.* We have implemented the RACE hashing and evaluated its performance. Extensive experimental results demonstrate that RACE hashing outperforms state-of-the-art distributed in-memory hashing indexes by up to 13.7× in YCSB [11] hybrid workloads.

2 BACKGROUND AND MOTIVATION

2.1 Disaggregated Memory

In a general disaggregated memory architecture in datacenters [6, 19, 30, 31, 44], different types of resources are separated into pools, e.g., a compute pool and a memory pool, as shown in Figure 1. Each pool is managed and scaled independently as well as failure isolated. The compute pool consists of many CPU blades, each of which retains a small amount of memory as the local cache for the memory pool. The memory pool includes many memory blades that can be DRAM or persistent memory DIMMs, RNICs, and controllers (the RNIC and controller can be the same entity). The RNIC and controller have low-power processing units used only for interconnection. The communication between compute and memory pools leverages fast remote-access interconnect techniques, such as one-sided RDMA, Omni-path [8], or Gen-Z [1]. The interfaces that the memory pool provides for the compute pool include `READ`, `WRITE`, `ALLOC`, and `FREE` for variable-size memory blocks, as well as `ATOMIC` operations, e.g., **compare-and-swap (CAS)** and fetch-and-add. We assume `ALLOC` and `FREE` interfaces are implemented in the RNICs or controllers of the memory pool [1, 44]. Without loss of generality, the rest of this article considers using

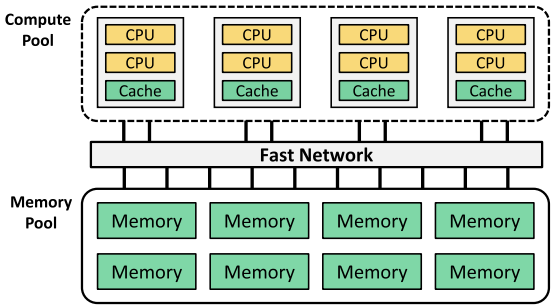

Fig. 1. Architecture for disaggregated memory.

one-sided RDMA for the interconnect in the disaggregated memory architecture, i.e., compute blades access the memory pool using RDMA `READ`, `WRITE`, and `ATOMIC` verbs.

2.2 RDMA-search-friendly Hashing Index

In this subsection, we first present existing RSF hashing indexes and then analyze their performance on disaggregated memory.

2.2.1 Existing Hashing Schemes. With the wide use of RDMA in modern datacenters, RSF hashing indexes have been intensively studied [14, 33, 46]. All these RSF hashing indexes are designed for the datacenter architecture with monolithic servers. Clients execute search requests by using RDMA `READ`s to fetch data from remote memory without involving remote CPUs. In contrast, IDU requests are sent to the remote servers and executed using the remote CPUs. We review each of these hashing indexes in detail as follows.

Pilaf Cuckoo Hashing: Pilaf [33] proposes a three-way cuckoo hashing that uses three orthogonal hash functions to compute three different hash buckets for each key. When executing a key `Search`, the client first reads one of its three corresponding hash buckets using an RDMA `READ`. If the key does not exist in the first bucket, then the client then reads the second hash bucket. Upon not finding the key in the second bucket, the client reads the third hash bucket. For `Insertion` requests, the client sends them to the server and the server CPUs handle them locally. An insertion may iteratively *evict* existing key-value items in the cuckoo hash table to their alternate locations. This mechanism incurs an inconsistency problem in which a search request executed by the client may miss the key when the server is handling its eviction. To address this problem, the server first calculates all affected buckets (called a cuckoo path [28]) before moving keys. The server then moves each key to its alternate location starting from the last affected bucket in the cuckoo path.

FaRM Hopscotch Hashing: FaRM [14] proposes a chained associative hopscotch hashing in which each bucket has a neighborhood that includes the bucket itself and its following bucket. Each bucket has multiple slots and each key is stored in the neighborhood of the bucket that the key is hashed to. For an `Insertion` that is also handled at the server side, the hopscotch hashing tries to find an empty slot in the neighborhood of the key's hash bucket. If found, then the empty slot stores the item. Otherwise, the hopscotch hashing continues to find an empty slot forward by executing a *linear probe*. If finding an empty slot, then the hopscotch hashing tries to iteratively displace items to *move* the empty slot toward the neighborhood. If there is no empty slot or the movement fails, then the hopscotch hashing stores the item in the bucket list linked to the key's hash bucket. When executing a `Search`, the client reads the neighborhood of the key's hash bucket, i.e., two adjacent buckets, using an RDMA `READ`. Upon not finding the key, the client further traverses the linked buckets. Note that traversing each bucket needs an RDMA `READ`.

DrTM Cluster Hashing: Cluster hashing proposed in DrTM [46] is a chained hashing with associativity, in which reading and writing key-value items use RDMA `READ`s and `WRITE`s and insertions and deletions to the hash table are shipped to the server for local execution. To `insert` a new key, the cluster hashing tries to find an empty slot in the key's hash bucket and the bucket list linked to the key's hash bucket. If there is no empty slot, then the cluster hashing adds a new bucket in the bucket list to store the inserted key-value item. For a `Search` request, the client reads the key's hash bucket using an RDMA `READ`. Upon not finding the key, the client further traverses the linked buckets one by one.

2.2.2 Performance on Disaggregated Memory. To the best of our knowledge, there is no existing hashing index specifically designed for disaggregated memory. As a first step, we analyze the performance of using the above RSF hashing indexes in disaggregated memory. Due to the absence of computing power in the memory pool to execute their IDU requests, we consider implementing the IDU requests with one-sided RDMA verbs.

For Pilaf cuckoo hashing [33], to insert a key-value item, the client needs to execute eviction operations when the hash table is in a high load factor. Specifically, based on the state-of-the-art concurrent cuckoo hashing algorithm [28], the client first calculates a cuckoo path and locks all buckets in the path using RDMA `CAS`es. The client then uses RDMA `WRITE`s to iteratively evict key-value items in the cuckoo path. A cuckoo path may include tens or hundreds of buckets [16]. Thus an insertion is executed by using a large number of RDMA `CAS`es and `WRITE`s, delivering poor insertion performance and also decreasing the performance of other search requests due to the use of a large number of locks.

For FaRM hopscotch hashing [14], to insert a key-value item, the client needs to linearly probe buckets in the hash table using RDMA `READ`s until finding an empty slot. When the hash table is in a high load factor, inserting a key may need to read the entire hash table to the client until finding an empty slot. After finding an empty slot, moving the empty slot toward the neighborhood of the inserted key is also complex and expensive, due to locking multiple buckets in the movement path and using multiple RDMA `WRITE`s to move items. Moreover, if there is no empty slot or the movement fails, then the operation of adding linked buckets is also expensive.

For DrTM cluster hashing [46], to insert a key-value item, the client needs to traverse buckets in its corresponding bucket list one by one until finding an empty slot. Traversing each bucket needs an RDMA `READ`. If there is an empty slot in these buckets, then the client inserts the item using an RDMA `WRITE`. Otherwise, the client adds a new overflow bucket to the bucket list. Before modifying the bucket list, the client needs to lock the bucket list to prevent other clients from inserting duplicate keys or freeing buckets. Thus an insertion executes operations including traversing the bucket list, locking/unlocking, allocating memory for a new bucket and the new item, linking the new bucket, and writing the new item, resulting in many RDMA `READ`s, `WRITE`s, and `CAS`es. The operations of allocating overflow buckets in the cluster hashing are more frequent than in FaRM hopscotch hashing, since the cluster hashing has a weaker ability to deal with hash collisions in the main hash table. Moreover, the deletion requests are also complex in the structure of linked bucket lists, due to the need of moving items from buckets at the list tail toward ones at the list head to fill empty slots and recycling tail buckets for higher performance and space utilization [14].

In summary, these RDMA-search-friendly hashing indexes become RDMA-IDU-unfriendly for disaggregated memory, since IDU requests incur a large number of RDMA operations to deal with hash collisions and concurrency control. Our article proposes RACE hashing that is *both RDMA-search-friendly and RDMA-IDU-friendly* while efficiently dealing with hash collisions and concurrency control as presented in Section 3. The performance is also verified in Section 5.

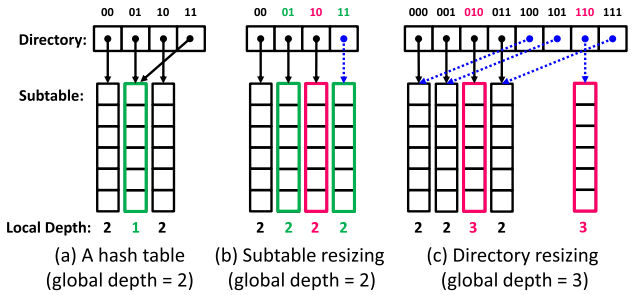

Fig. 2. A hash table with extendible resizing.

2.3 Resizing Hash Tables

When a hash table is full, i.e., an insertion failure occurs or its load factor reaches a threshold, the hash table needs to be resized by expanding its capacity. In general, there are two kinds of resizing mechanisms including full-table resizing and extendible resizing [15, 35].

To expand a hash table, the *full-table resizing* mechanism allocates a new hash table whose size is larger than the old one, e.g., double the size, and then iteratively moves each key-value item from the old hash table to the new one. The full-table resizing is expensive due to moving all items.

In the *extendible resizing*, a resizing operation only needs to move partial items. Specifically, the hash table using extendible resizing includes multiple subtables, and there is a directory to index these subtables as shown in Figure 2(a). For a 64-bit hash value, M bits are used by the directory to locate a subtable (we use the last M bits as an example, i.e., suffix) and the remaining $(64 - M)$ bits are used to locate target buckets within the subtable. The number of suffix bits currently used by the directory is called ***global depth (GD)*** ($GD \leq M$). Thus the directory has 2^{GD} entries that correspond to at most 2^{GD} subtables. Each subtable has a ***local depth (LD)*** ($LD \leq GD$) that indicates the number of suffix bits used by the subtable.

When a subtable is full, we split the subtable into two by adding a new subtable. As shown in Figure 2(a) and (b), when the subtable with the suffix "1" is full, it is split into Subtables "01" and "11." The resizing mechanism moves the key with suffix "11" from Subtable "01" to Subtable "11" and changes their *LDs* to 2. When a subtable is full and its *LD* is equal to the *GD*, we grow the directory by doubling its size, as shown in Figure 2(b) and (c). The full subtable is split into two ones. Except for the directory entry that the added new subtable corresponds to, other new directory entries point to their corresponding original subtables. After resizing the directory, search requests use the new *GD* to locate their corresponding subtables. In summary, by performing extendible resizing, when a subtable is full, we only need to resize this single subtable without affecting key-value items in other subtables.Therefore, RACE hashing uses the extendible resizing.

Nevertheless, there are two challenges when using extendible resizing in disaggregated memory. First, compared with the full-table resizing, the extendible resizing incurs one extra memory access for each search request, due to the need of first querying the directory to obtain the address of the target subtable before accessing the subtable. One extra memory access has little impact on the performance of a local hash table, due to fast local memory access. However, in the disaggregated memory, the one extra memory access *produces one more RDMA round-trip*, significantly decreasing the search performance. Second, as there is no powerful compute resource in the disaggregated memory to execute the complex resizing, the resizing has to be triggered and executed by a remote client, i.e., a CPU blade, which is different from the traditional resizing mechanism that is always executed by local CPUs. When a client is performing the resizing, other clients do not know about its occurrence. Therefore, we have to *deal with concurrent access to the hash table during resizing*.

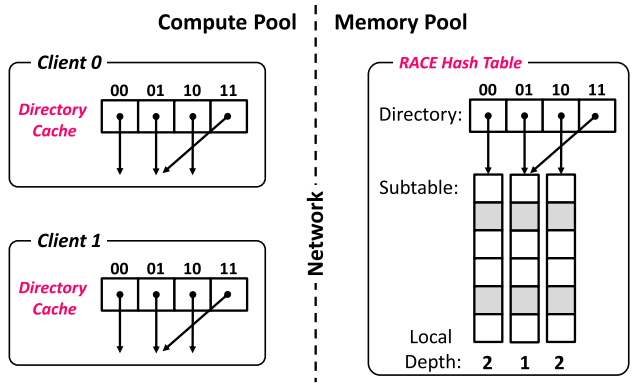

Fig. 3. The overall architecture of RACE hashing in disaggregated memory. The entire RACE hash table is stored in the memory pool. Clients in the compute pool store the directory of the hash table in their local caches and access subtables only using one-sided RDMA verbs.

3 RACE HASHING

3.1 Overview

Figure 3 shows the overall architecture of RACE hashing for disaggregated memory. The RACE hash table is stored in the memory pool. Clients in the compute pool operate the hash table using one-sided RDMA verbs. To alleviate the performance influence of resizing, RACE hashing leverages the extendible resizing and hence the hash table consists of multiple subtables and a directory. To reduce the extra RDMA `READ` for accessing the remote directory, RACE hashing leverages a directory cache[1] in the client. Each client maintains a local cache to store only the directory of the RACE hash table. Thus a client can access the directory using a local memory access rather than a remote RDMA `READ`, and use RDMA verbs to access only the subtable. We present the design of *an RAC hash subtable structure* in Section 3.2, i.e., the RAC hash subtable, in which all index requests are executed by using only one-sided RDMA verbs while having constant worst-case time complexity. We then present *a lock-free remote concurrency control scheme* in Section 3.3 for the RAC hash subtable, achieving that index requests including search, insertion, deletion, and update are concurrently executed in a lock-free way. Moreover, caching the directory in clients causes data inconsistency issues between the directories in the memory pool and client caches. Therefore, we finally present *a client directory cache with stale reads scheme* in Section 3.4 to address the inconsistency issue at low overhead.

3.2 The RAC Hash Subtable Structure

In disaggregated memory scenarios, the challenge of designing a RAC hash subtable structure stems from minimizing the number of remote RDMA operations for IDU requests while keeping high memory efficiency and `Search` performance. To achieve this goal, we design the RAC hash subtable that does not allow any movement operations, evictions, or bucket chaining to handle hash collisions, since these operations incur a large number of remote writes as presented in Section 2.2. Instead, the RAC hash subtable uses three major design choices, including associativity, two choices, and overflow colocation, for addressing hash collisions and thus achieves a constant worst-case time complexity for all index requests.

[1]The memory overhead of the cache is small, since the directory generally has at most hundreds of entries and each entry has only several bytes.

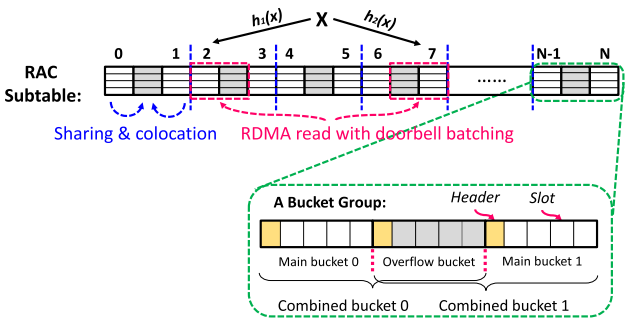

Fig. 4. The RAC hash subtable structure (four-way associativity as an example).

D1: Associativity. With associativity, each bucket has multiple slots, being capable of storing multiple key-value items. K-way associativity means that each bucket has K slots. Associativity is friendly for one-sided RDMA operations, since multiple items within one bucket can be read together in one RDMA READ. Figure 4 shows a RAC hash subtable with four-way associativity.

D2: Two Choices. Based on the theory of "the power of two choices" [34], enabling each key to have two choices for its storage location can achieve a good load balance among buckets, effectively handling hash collisions. Hence, the RAC subtable uses two independent hash functions, $h_1()$ and $h_2()$, to compute two hash locations for each key, as shown in Figure 4. By efficiently combining associativity with two choices, the RAC subtable inserts a new item into the less-loaded bucket between its two hash locations. As shown in Figure 5, when both associativity and the technique of two choices are applied, the maximum load factor of RACE hash table is significantly improved. Note that, according to Mitzenmacher's observations [34], two choices achieve exponential improvements over one choice for the efficiency of load balancing, while three choices only have a constant factor improvement than two choices. In disaggregated memory, three choices incur one more bucket access (i.e., one more RDMA READ) than two choices. Therefore, unlike Pilaf [33], which uses three choices, we use two choices in our design.

D3: Overflow Colocation. The overflow sharing technique [48] enables an overflow bucket (or called standby bucket) to be shared by the other two main buckets to store conflicting items for better load balancing, which can further improve the load factor of RACE hash table as shown by D1+D2+D3 in Figure 5. However, overflow buckets are discrete from their main buckets [48], incurring extra bucket accesses, which performs worse for disaggregated memory, due to extra RDMA READs. To address this problem, we propose *an overflow colocation scheme* to store the overflow buckets adjoining with their main buckets. As shown in Figure 4, three continuous buckets are considered as a group, in which the first and last buckets are main buckets that can be addressable by the hash functions. The middle bucket is an overflow bucket that cannot be addressable by the hash functions and is shared by the first and last buckets to store their conflicting items. By doing so, one RDMA READ can fetch one main bucket and its overflow bucket together, thus reducing the number of RDMA READs.

Putting it all together, the structure of a RAC hash table is shown in Figure 4. A RAC hash subtable is a one-dimensional bucket array stored in a continuous memory space. Each bucket is K-way associative and a bucket group includes three continuous buckets, i.e., two main buckets and a shared overflow bucket. The combination of a main bucket and its overflow bucket is called *a combined bucket*. For each key, we compute two hash locations that are respectively in two different bucket groups. The structure of the RAC hash subtable is simple yet efficient for disaggregated memory, having the following strengths:

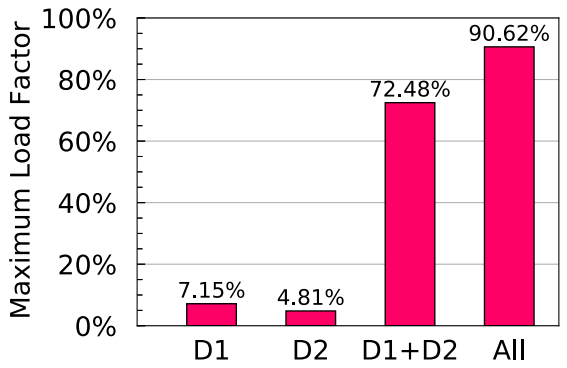

Fig. 5. The maximum load factors with different design choices. The maximum load factor is defined as the current load factor of the hash table when an insertion failure occurs. D1: RACE hash table with seven slots per bucket; D2: RACE hash table with Two Choices; D1 + D2: RACE hash table with D1 and D2; All: RACE hash table with D1, D2, and D3.

• *RDMA-IDU friendly:* As each key only involves two combined buckets, IDU requests only need to operate within the two combined buckets without moving/evicting items from/to other buckets or linking new buckets, having constant worst-case time complexity while being RDMA-friendly.

• *RDMA-search friendly:* A search request only issues two RDMA READs, each of which fetches one combined bucket. More importantly, the two RDMA READs can be issued in parallel to reduce the request latency, unlike cluster hashing [46] in which issuing the next RDMA READ has to wait for the return of the previous one to traverse the linked buckets. Moreover, by using doorbell batching [24] that is an RDMA-optimized technique to read multiple disjoint memory regions within one RDMA **round-trip time (RTT)**, we package the two RDMA READ operations into one. Therefore, the search latency in the RAC subtable is one RTT rather than two ones.

• *High memory efficiency:* By combining associativity, two choices, and overflow colocation, the RAC hash subtable enables items to be more evenly distributed among buckets and thus efficiently handles hash collisions to achieve a good load balance. It hence achieves a high load factor of up to 90% (with seven-way associativity) as evaluated in Section 5.2.1.

3.3 Lock-free Remote Concurrency Control

Lock-based techniques have been widely used in existing hashing indexes within a single machine for concurrency control [16, 28]. Nevertheless, for disaggregated memory, all requests are executed by using one-sided RDMA verbs, which results in non-trivial challenges for handling concurrent access conflicts. This is because remote locking implemented by using microsecond-level-latency RDMA CAS incurs much higher overheads, compared with nanosecond-level-latency local locking, and each locking or unlocking operation requires an RDMA round-trip. To deliver high concurrent performance, we propose *a lock-free remote concurrency control scheme* for RACE hashing that achieves that all index requests, except failed insertions, become lock-free. A failed insertion triggers a subtable resizing and needs to acquire the resizing lock as presented in Section 3.4.2.

Bucket Structure. In RACE hashing, to support variable-length keys and values, full key-value items are stored outside the hash table like existing hashing indexes [10, 33, 46]. The pointers to full key-value items are stored inside the hash table. The structure of each bucket in the RAC hash subtable is shown in Figure 6. A bucket consists of a header and multiple slots. The header is used for hash table resizing and will be introduced in Section 3.4.1. Each slot corresponds to a

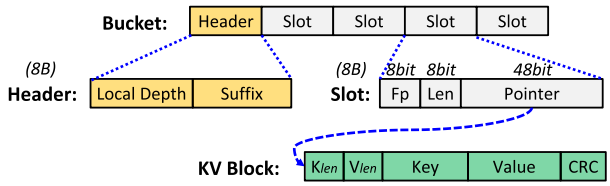

Fig. 6. The structure of a bucket. (Fp: The fingerprint of the key, which is a 8-bit hash of the key. Len: The length of the key-value block in the slot.)

key-value item. To support lock-free remote concurrent access, a slot is 8 B, i.e., the maximum size of an RDMA `CAS`, and composed of a fingerprint (8 bits), a key-value length (8 bits), and a pointer (48 bits). A `fingerprint` is the 8-bit hash of a key. Based on the analysis of existing work [16], an 8-bit fingerprint is enough to achieve a very low false positive (a false positive means that different keys in a bucket have the same fingerprint). Moreover, before reading a full key-value item using an RDMA `READ`, we need to know the size of the item. Therefore, we store the `length` of the key-value block in the slot. The length is 8-bit and the length unit is 64 B.[2] Thus the length of a key-value block is always multiple of 64 B and the maximum length of a key-value block is $2^8 * 64$ B = 16 KB, which covers most application scenarios for current key-value stores, since small key-values dominate in them [7]. When a key-value item is larger than 16 KB, which in fact rarely occurs, we store the remaining item content beyond 16 KB in the second key-value block and link the second block to the first one. The respective lengths of the key and value (i.e., K_{len} and V_{len}) are stored in the head of the key-value block. The `pointer` in a slot consumes 48 bits like the x86_64 system [10, 36, 45]. A null pointer means the slot is empty. Based on the bucket structure, we present lock-free search/insertion/deletion/update operations below.

Lock-free Insertion. To insert a key-value item, the client uses doorbell batching [24] to read two combined buckets that the key corresponds to. At the same time, the client writes the key-value block[3] in the memory pool. Therefore, reading buckets and writing the key-value block are executed in parallel, as shown in Figure 7(b). Once receiving two combined buckets, the client first chooses the less-loaded one. The client then looks for an empty slot in the order of main buckets first and overflow buckets second, as presented in Section 3.2. If an empty slot is found, then the client uses an RDMA `CAS` to write the pointer of the key-value block into it. Otherwise, the hash table resizing is triggered, as presented in Section 3.4.

In rare cases, clients may concurrently insert duplicate keys into the hash table, since RDMA `ATOMIC` verbs only ensure the 8-B atomicity. For example, Client 1 and Client 2 try to insert the same key K. As each key corresponds to two combined buckets in RACE hashing, it may occur that Client 1 selects one empty slot in Combined Bucket 0 to insert K, and Client 2 selects one empty slot in Combined Bucket 1 to insert K. In this case, two duplicate keys K exist in the hash table. To address the issue of duplicate keys, after writing the pointer in a bucket for an insertion, the client re-reads the two combined buckets to check duplicate keys, as shown in Figure 7(b). On finding duplicate keys, the client only keeps one *valid* key and removes the remaining duplicate keys. Different clients have to determine the same key-value item as the valid key when finding duplicate keys to guarantee the consistency of a concurrent access. To guarantee this, we hence make an agreement in the algorithm to determine the only valid key for different clients. For

[2]The length unit can be changed as needed.
[3]The memory of the key-value block can be pre-allocated to reduce the latency of memory allocation in the critical path of insertion, the detail of which is shown in Section 4.2.

ALGORITHM 1: Insert (key, value)

```
// 1st RTT: Using RDMA doorbell batching to fetch two combined buckets
// and write the new key-value pair asynchronously
suffix = ExtractSuffix(key, directory.globalDepth)
subtable = directory.subtable[suffix]
index_1 = Hash_1(key)
index_2 = Hash_2(key)
bucket_1, bucket_2 = RdmaRead(subtable[index_1], subtable[index_2])
remoteAddress = AllocBlock()
RdmaWrite(key, value, remoteAddress)
// If the subtable is being resized, restart the insertion
if (subtable.localDepth != bucket1.localDepth || subtable.localDepth != bucket2.localDepth)
    Synchronize(directory)
    restart
bucket = FindLessLoadBucket(bucket_1, bucket_2)
slot = FindEmptySlot(bucket)
// If no empty slot is found, the hash table resizing is triggered
if (slot == NULL)
    Resize()
    restart
// 2nd RTT: Using RDMA CAS to write the pointer of the key-value block
entry.fingerprint = key.fingerprint
entry.pointer = remoteAddress
if (RdmaCas(slot, entry) != SUCCESS)
    restart
// 3rd RTT: Re-reading two combined buckets to remove duplicate keys
bucket_1, bucket_2 = RdmaRead(subtable[index_1], subtable[index_2])
RemoveDuplicate(bucket_1, bucket_2, key)
// If the cache is stale, remove the key-value pair and restart the insertion
if (IsBucketCorrect(subtable, bucket, key) == false)
    RemoveKey(key, slot)
    Synchronize(directory)
    restart
return SUCCESS
```

example, within the two combined buckets, the agreement considers the key stored in the slot with the minimal bucket number and the minimal slot number to be the only valid one. The pseudo-code of the lock-free insertion operation is shown in Algorithm 1.

Lock-free Deletion. To delete a key-value item, the client first executes a search to find the target key. If the target key is found, then the client sets its corresponding slot to be null by using an RDMA CAS, as shown in Figure 7(c). Once the RDMA CAS is done successfully, the deletion request is returned. The client then sets the key-value block to full-zero and frees the key-value block in background. The zero-setting operation can be avoided if the RNIC can automatically set the freed memory to full-zero for data security, i.e., avoiding the old data to be observed by other clients. The pseudo-code of the lock-free deletion operation is shown in Algorithm 2. In rare cases, a client may find duplicate keys during a deletion, which is caused by an on-going insertion. In this case, the client deletes all duplicate keys.

Lock-free Update. To update a key-value item, the client searches the target key. At the same time, the client writes the new key-value item into the memory pool, as shown in Figure 7(d). Once finding the target key exists, the client uses an RDMA CAS to change the content of the slot to point

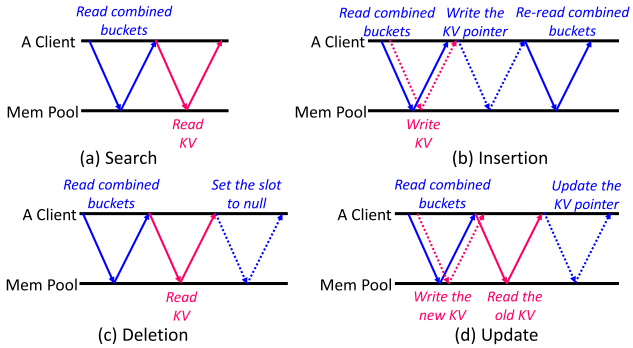

Fig. 7. The main workflows of lock-free search, insertion, deletion, and update. The blue lines mean accessing the hash table, and the red lines mean accessing the key-value blocks. The solid lines mean RDMA READ round-trips, and the dotted lines mean RDMA WRITE/ATOMIC round-trips.

ALGORITHM 2: DELETE (key)

```
// 1st + 2nd RTT: Searching for the subtable, bucket and slot that contains the key
// The implementation of the function Search'() is similar to the function Search()
// while it returns a subtable, a bucket, and a slot instead of a value
subtable, bucket, slot = Search'(key)
// If the cache is stale, restart the deletion
if (IsBucketCorrect(subtable, bucket, key) == false)
    restart
if (slot != NULL)
    // 3rd RTT: Setting the key-value block to full zero
    // and freeing the key-value block asynchronously in the background
    if (RdmaCas(slot, NULL) == SUCCESS)
        Reset(slot.pointer)
        Free(slot.pointer)
    else
        restart
return SUCCESS
```

to the new key-value item. If the RDMA CAS is executed successfully, then the update request is returned. The client finally frees the old key-value block in background. The pseudo-code of the lock-free update operation is shown in Algorithm 3.

Lock-free Search. As shown in Figure 7(a), to search a key, the client reads its corresponding two combined buckets. If the fingerprint matches one of the slots, then the client reads the key-value block that the slot points to. The client then compares the full key stored in the key-value block. If the full key matches, then the value is returned.

Since all modifications on buckets are atomic and update requests do not modify the old key-value item in place, the only inconsistency case for a search is that the key-value block is freed or re-allocated before a search request reads the key-value block (after obtaining the pointer of the key-value). However, this inconsistency case can be easily observed by comparing the length and content of the key stored in the block with those of the search key. This is because once the key-value block is freed/re-allocated, its content is full-zero/changed, rendering the comparison mismatched. Nevertheless, there still exists a special case that another client re-allocates the key-value block and issues an RDMA WRITE to write the same key, key length, and value length

ALGORITHM 3: Update (key, value)

```
// 1st + 2nd RTT: Writing the new key-value block and searching for the subtable, bucket and slot
// that contains the key at the same time
remoteAddress = AllocBlock()
RdmaWrite(key, value, remoteAddress)
subtable, bucket, slot = Search'(key)
// If the cache is stale, restart the update
if (IsBucketCorrect(subtable, bucket, key) == false)
    restart
if (slot != NULL)
    entry.fingerprint = key.fingerprint
    entry.pointer = remoteAddress
    // 3rd RTT: Updating the content of the slot
    // and freeing the old key-value block asynchronously in the background
    if (RdmaCas(slot, entry) == SUCCESS)
        Reset(slot.pointer)
        Free(slot.pointer)
    else
        restart
return SUCCESS
```

ALGORITHM 4: Search (key)

```
// 1st RTT: Using RDMA doorbell batching to fetch two combined buckets
suffix = ExtractSuffix(key, directory.globalDepth)
subtable = directory.subtable[suffix]
if (subtable.lock == true)
    return SearchOnResize(key)
index1 = Hash1(key)
index2 = Hash2(key)
bucket1, bucket2 = RdmaRead(subtable[index1], subtable[index2])
// If the cache is stale, further search is needed and the detail is shown in Section 3.4.2
if (IsBucketCorrect(subtable, bucket1, key) == false ||
    IsBucketCorrect(subtable, bucket2, key) == false)
    return SearchOnResize(key)
// Search the slots of two buckets for the key
returnValue = NULL
for each slot in bucket1 and bucket2
    if (slot.footprint == key.footprint)
        // 2nd RTT: Reading the key-value block of the current slot
        block = RdmaRead(slot.pointer)
        // Check the full key in the block and the checksum of the block
        if (block.key == key && Check(block.crc) == SUCCESS)
            returnValue = kvblock.value
            break
return returnValue
```

as those of the old key-value block. As an RDMA `WRITE` is not atomic, it may write the key and key length completely but be writing the value. At this time, if reading the key-value block, then a client can find the key is matched. But the value is broken, which cannot be observed by the client. To address this problem, we add a 64-bit checksum in each key-value block to enhance the self-verification and check the integrity of a key-value block like Pilaf [33], as shown in Figure 6.

Pilaf also shows that a 64-bit checksum is sufficient for verification. The pseudo-code of the lock-free search operation is shown in Algorithm 4.

Moreover, for insertion, deletion, and update requests, performing a CAS operation on a slot may fail, which means the slot is changed by another client before the CAS operation. In this case, RACE hashing re-searches the target key and then re-executes the failed insertion, deletion, or update request.

3.4 Extendible Remote Resizing

Using extendible resizing for disaggregated memory incurs two challenges, i.e., one extra remote access to read the directory for each index request and concurrent access during resizing, as presented in Section 2.3. In this subsection, we present a stale-read client directory cache scheme and a concurrent access scheme during resizing to address the two challenges, respectively.

3.4.1 Client Directory Cache with Stale Reads. To reduce the extra RDMA READ for accessing the directory, we use a client directory cache for RACE hashing. However, caching the directory in clients incurs the data inconsistency issue between the directories in the memory pool and client caches. For example, when a client triggers a subtable resizing or directory resizing, the content of the directory in the memory pool is modified, and thus the directories in the caches of other clients become stale. If other clients still query the hash table using their stale directories, then they may locate an incorrect subtable and obtain incorrect data.

To address the inconsistency problem between client caches and the memory pool, in *a baseline solution* [18], and upon a client triggers a resizing operation, the client broadcasts a notification message to all other clients to invalidate their respective directory caches and does not start modifying the directory in the memory pool until receiving acks of all other clients. Obviously, the baseline solution incurs high performance overhead for resizing due to broadcasting messages and waiting for all acks. *The second solution* proposed by Pilaf [33] is to close the RDMA connections of all other clients to prevent these clients from performing RDMA READs once a resizing is triggered. Clients then re-connects the memory server to obtain the new table root after the resizing is completed. Pilaf addresses the problem of incorrect access but incurs high performance penalty due to blocking RDMA READs of clients. Therefore, both the solutions incur significant performance overheads.

To efficiently address this inconsistency problem, we propose a **stale-read client directory (SRCD)** cache scheme that does not need to broadcast messages or close the connections of other clients to the memory pool when triggering a resizing. Instead, by using the SRCD cache scheme, clients query the hash table still using the stale directories in their caches but can verify whether the obtained data are correct. To achieve this, we add a header in each bucket of the RAC hash subtable, as shown in Figure 6. The bucket header stores the LD and `suffix bits` (Suffix) of the subtable that the bucket belongs to. The bucket header is not modified in IDU requests and is modified only when the subtable is created and resized. The `local depth` and `suffix bits` in the bucket header are used to verify whether the bucket is correct when executing search/insertion/deletion/update.

Figure 8 shows an illustration of using the SRCD cache scheme to verify the correctness of buckets. The client currently caches the directory of the hash table shown in Figure 2(a), in which the directory entries "01" and "11" point to the same subtable. In the memory pool, the hash table is resized to be a new hash table shown in Figure 2(b), in which a new subtable is created and the directory entries "01" and "11" point to different subtables. To search a key, the client first locates a subtable using the SRCD cache with stale reads and then fetches the buckets in this subtable via RDMA READs. After receiving a bucket, the client respectively compares the `local depth` and

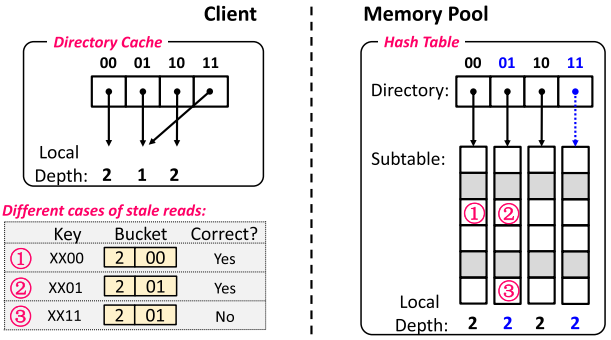

Fig. 8. The stale-read client directory scheme. Three cases when comparing the key with the fetched bucket header.

`suffix bits` stored in the bucket header with the `local depth` of the directory entry in the SRCD cache and the `suffix bits` of the key. The client can observe three cases as follows.

(1) Both local depth and suffix bits match. If the local depth and suffix bits in the bucket header are respectively the same as the local depth of the directory entry in the cache and suffix bits of the key, then the client can verify that the subtable is not resized and the fetched bucket is correct. For example, the key is "XX00" that corresponds to the directory entry "00," i.e., Case ① in Figure 8.

(2) Local depth mismatches and suffix bits matches. If the local depth in the bucket header is the same as that of the directory entry in the cache, then the client knows the accessed subtable was resized in the memory pool. The client further computes the suffix bits of the key using the local depth stored in the bucket header and finds that the suffix bits of the key and those stored in the bucket header are matched. In this case, the client can verify the bucket is also correct although the subtable was resized. For example, the key is "XX01" but Subtable "01" was resized in the memory pool, i.e., Case ② in Figure 8. During the resizing, the keys with "11" are moved out Subtable "01" while the keys with "01" still remain in Subtable "01." Therefore, when locating Subtable "01" to search the key with "01," the client can obtain the correct key-value item.

(3) Both local depth and suffix bits mismatch. If the local depth and suffix bits in the bucket header mismatch, then the client can verify the subtable is resized and the searched key should be stored in the new subtable. For example, the key is "XX11" that corresponds to the directory entry "01" in the cache but Subtable "01" is resized in the memory pool and the key "XX11" is moved to the new Subtable "11," i.e., Case ③ in Figure 8. In this case, the client fetches the new directory entries from the memory pool and re-executes the search.

In summary, only in Case "*3) Both local depth and suffix bits mismatch,*" the client needs to fetch new directory entries and update the local directory cache. In other cases, the client can locate correct buckets via stale reads. The pseudo-code of verifying the correctness of buckets is shown in Algorithm 5.

3.4.2 Concurrent Access during Resizing. When an insertion failure occurs, a subtable resizing is triggered. During a resizing, we need to move slots from the resized subtable to the new one. Due to the slot movement, it is challenging to guarantee the correct execution of concurrent search, insertion, deletion, and update requests upon the subtable that is being resized. To address this challenge, we design the workflow of concurrent resizing as below.

To support concurrent access during resizing, the starting address of the directory in the memory pool cannot be changed. Otherwise, clients fail to find the new hash table after resizing.

ALGORITHM 5: IsCorrectBucket (subtable, bucket, key)

```
// Compare the local depth of the bucket and the subtable entry to detect resizing
if (bucket.localDepth != subtable.localDepth)
    // Compare the suffix of the bucket and the suffix of the key to detect cache staleness
    suffix = ExtractSuffix(key, bucket.localDepth)
    if (suffix != bucket.suffix)
        return false
return true
```

ALGORITHM 6: SearchOnResize (key)

```
// Synchronize the cache and read two corresponding combined buckets
Synchronize(directory)
suffix = ExtractSuffix(key, directory.globalDepth)
subtable = directory.subtable[suffix]
index1 = Hash1(key)
index2 = Hash2(key)
bucket1, bucket2 = RdmaRead(subtable[index1], subtable[index2])
// Search the slots of two buckets for the key
for each slot in bucket1 and bucket2
    if (slot.footprint == key.footprint)
        // Read the key-value block of the current slot
        block = RdmaRead(slot.pointer)
        if (block.key == key && Check(block.crc) == SUCCESS)
            return block.value
// Detect cache staleness
if (IsBucketCorrect(subtable, bucket1, key) == false ||
    IsBucketCorrect(subtable, bucket2, key) == false)
    restart
// Check if the subtable is being resized
returnValue = NULL
if (subtable.lock == true && suffix.firstBit == 1)
    // Search in the old subtable before resizing
    oldSuffix = ComputeOldSuffix(suffix)
    oldSubtable = directory.subtable[oldSuffix]
    bucket1, bucket2 = RdmaRead(oldSubtable[index1], oldSubtable[index2])
    returnValue = SearchBucket(bucket1, bucket2)
    // Search in the subtable again
    if (returnValue == NULL)
        bucket1, bucket2 = RdmaRead(subtable[index1], subtable[index2])
        returnValue = SearchBucket(bucket1, bucket2)
return returnValue
```

Therefore, we reserve a large enough contiguous memory space[4] used for the future resizing of the directory. As shown in Figure 9, the directory includes a used area and an unused area. Clients only cache the used area. To resize the directory, e.g., increasing the GD from 1 to 2, the used area is not changed and the new directory entries are written into the unused area.

To resize a subtable, the client first locks the directory entry of the subtable in the memory pool. The lock only prevents other clients from resizing the same subtable but does not prevent other

[4]For example, if we use at most 16 suffix bits for the directory, then the memory space of 2^{16} directory entries is reserved.

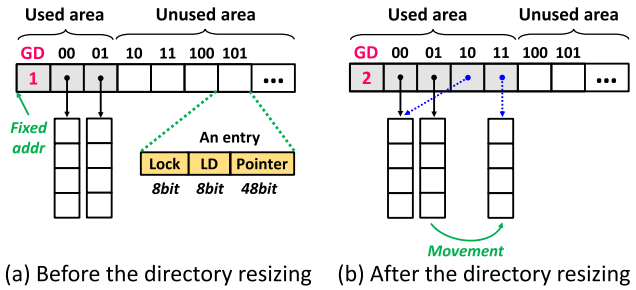

Fig. 9. The resizing of the directory. The GD increases. The used area is not changed and new directory entries are written into the unused area.

clients from executing search and IDU requests in the subtable. The client creates a new subtable and initializes the header of each bucket in the new subtable. The client further writes the pointer of the new subtable to the directory and locks the directory entry at the same time. The client then starts to move items. Figure 9(b) shows an example. The client moves items with Suffix "11" from Subtable "01" to Subtable "11." The movement includes three steps for each bucket in Subtable "01": ❶ updating the suffix bits in the bucket header from "1" to "2" (one RDMA CAS), ❷ inserting all items with Suffix "11" in this bucket into Subtable "11" (one or multiple RDMA CASes), and ❸ deleting all items with Suffix "11" in this bucket (one or multiple RDMA CASes). By guaranteeing the order of executing the three steps, we support concurrent access to the subtable that is being resized. In the following, we discuss how to deal with the corner cases caused by the concurrent resizing below.

• *Concurrent search:* As shown in Algorithm 6, when executing a search, if we find that both local depth and suffix bits mismatch, then the client can perceive the occurrence of the resizing in the current subtable. In this case, the movement may be before Step ❸ or after Step ❸. If the target key is found in the read bucket, then it means that the movement is before Step ❸ and the search is complete. Otherwise, the movement is after Step ❸ or the key does not exist, in which case the search operation will be re-executed by the client. If we find that suffix bits match but the local depth mismatches, then the client has read the correct buckets. However, it is possible that the target key has not been moved to the newly created subtable yet. The client further reads the corresponding buckets in the old subtable to search for the target key. If the target key is not found in the old subtable, then it is possible that the target key has been moved to the newly created subtable. The client re-reads the buckets in the new subtable to complete the search.

• *Concurrent insertion:* During an insertion, the client re-reads the buckets to check duplicate keys, as shown in Figure 7(b). To support concurrent insertion during resizing, we also check the bucket header after re-reading the bucket that the key is inserted to. If the bucket header is not changed, then the insertion is successful. Otherwise, the client knows that a resizing occurs in the bucket. The client then compares suffix bits in the new bucket header with those of the inserted key. If the suffix bits match, then the insertion is also successful. Otherwise, the client removes the pointer from Subtable "01" and re-inserts the key into Subtable "11." Moreover, during a subtable resizing, an insertion may fail, i.e., not finding an empty slot in the subtable. In this case, the failed insertion triggers the next resizing. The next resizing will be blocked until the previous resizing releases the directory entry lock.

• *Concurrent deletion/update:* When executing a deletion/update, if finding that both local depth and suffix bits mismatch, then the client waits for the completion of the movement and then deletes/updates the key from the new subtable. If the suffix bits match and Step ❶ occurs before the

RDMA CAS operation of the deletion/update, then there are two corner cases. First, if the RDMA CAS of the deletion/update fails, then it means the item has been moved into the new subtable. The client will redo the deletion/update request in the new subtable. Second, the RDMA CAS of the deletion/update succeeds. But the client performing resizing operation fails to delete one item in Step ❸, which means that another client deleted/updated the item. The client performing resizing further cleans the pointer of the item in the new subtable and re-executes the item movement.

In summary, during a subtable resizing, all search/update/ deletion and most insertion requests to the subtable are concurrently executed in a lock-free way. Only the failed insertions to the subtable are blocked due to triggering the next resizing.

4 IMPLEMENTATION DETAILS

In this section, we first present how to use the co-routine technique [13, 20] to improve the client-side request throughput of RACE hashing. We then present the implementation of the pre-allocation mechanism, which is used to reduce the overhead of remote memory allocation.

4.1 Client-side Request Handling

As mentioned in Section 3.1, the RACE hash table is stored in the memory pool and each client process running on a compute node operates the hash table using one-side RDMA verbs. Within a compute node, different applications send requests to the RACE client process to execute corresponding operations (i.e., search, insertion, update, and deletion). A naive solution for a client process to handle concurrent requests is to push them into a message queue and deal with them one by one. However, such a naive solution cannot achieve good performance due to the high execution latency. Specifically, each RACE operation contains multiple RDMA READs and WRITEs. If a client process handles requests sequentially, then the CPU core becomes idle during each RDMA READ and WRITE, resulting in a waste of CPU resources and low overall throughput.

To deal with the problem mentioned above, we leverage the co-routine technique [13, 20] to improve the efficiency of RACE hashing client. Co-routines are special functions that can be suspended and resumed at manually defined points during executions. In contrast to threads, the switches of which are decided by the operating system, co-routine switches are controlled by the programmers and do not involve any system calls.

We implement each operation of RACE hashing as a co-routine function and each function is further decomposed into several consecutive steps. A step is a code piece of an operation that contains exactly one RDMA verb or return statement at the end. As shown in Figure 10, the search operation is divided into three steps. In the first step, the search opertion computes the addresses of the buckets and launches an RDMA READ to fetch the buckets. In the second step, the search operation searches in the read buckets for the target key and lanuches an RDMA READ to fetch the corresponding key-value block. In the third step, the search operation compares the whole key stored in the key-value block with the target key and returns the value.

Different from the naive solution, where operations are executed sequentially, a RACE client executes concurrent co-routine functions at the granularity of steps. To be specific, when receiving multiple requests, a RACE client process invokes a batch of co-routine functions to deal with those requests concurrently. Figure 11 shows an example where two concurrent searches are executed by a client process running with one thread. When the client process reaches the end of a step, it launches the corresponding RDMA verb asynchronously and then switches to a ready-to-execute step of another concurrent co-routine function. The scheduling policy of those concurrent co-routine functions is decided by the underlying infrastructure. By taking advantage of co-routines, RACE hashing manages to overlap CPU computation and network transmission among concurrent operations, thus improving CPU resource utilization as well as achieving high throughput.

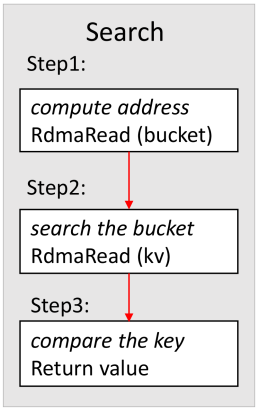

Fig. 10. Steps of a search.

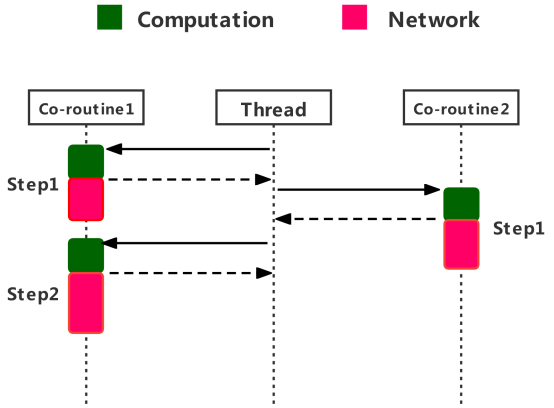

Fig. 11. Two concurrent searches executed by one thread.

4.2 Memory Pre-allocation

As mentioned in Section 2.1, processes running in the compute pool leverage the interfaces provided by the memory pool to allocate and free variable-size memory blocks from the memory pool. Since current RNIC hardware does not support allocating or freeing remote memory, we implement the interfaces of memory allocation and deallocation with **remote procedure call (RPC)**. In addition, to emulate the weak compute power of the memory pool, we only use one thread on each machine in the memory pool to deal with memory allocations and deallocations.

In RACE hashing, a client needs to allocate memory for key-value blocks in insertion and update operations to store new key-value pairs. To allocate a key-value block from the memory pool, a RACE client launches an RPC to the memory pool and waits for the remote memory address of the key-value block. If a client only allocates one key-value block in each RPC, then a network round-trip is added to the critical paths of these operations and the latency is increased. What makes things worse is the fact that we need to rely on the weak computing power of the memory pool to deal with RPCs, which further exacerbates the performance of insertion and update operations.

To address the problem mentioned above, RACE hashing applies a pre-allocation mechanism in the current implementation. To be specific, each RACE client maintains a local queue on the compute node to store remote addresses of all allocated key-value blocks and the local queue is managed by a background thread. When the number of free key-value blocks in the local queue is smaller than a lower limit, the background thread invokes an RPC to allocate memory for a batch of key-value blocks from the memory pool. To avoid assigning too much memory to a client, the background thread also invokes RPCs to release some memory back to the memory pool when the size of the local queue is larger than an upper limit. A RACE client allocates key-value blocks directly from the queue and frees key-value blocks by putting the addresses of those blocks back to the local queue. With the help of this pre-allocation mechanism, RACE hashing removes memory allocations out of the critical paths of insertion and update operations, reducing the latency and releasing the burden of the weak computing units in the memory pool.

5 PERFORMANCE EVALUATION

5.1 Experimental Setup

Testbed. We run all experiments on five machines, each with two 26-core Intel Xeon Gold 6278C CPUs, 384 GiB DRAM, and one 100-Gbps Mellanox ConnectX-5 IB RNIC. Each RNIC is connected

to a 100-Gbps Mellanox IB switch. One machine is used for emulating the memory pool. As mentioned in Section 4.2, to emulate the weak compute power, we run a process with only one thread in the memory pool to deal with memory allocation/deallocation requests. The memory is allocated and registered with huge pages to reduce the page translation cache misses of RNICs [14]. Other machines are used for building the compute pool in which each CPU core serves as a client.

Workloads. We use YCSB [11] to evaluate the performance of different hashing indexes. We use the default Zipfian request distribution ($\theta = 0.99$) for all YCSB workloads. For most experiments, we use 16-byte keys and 32-byte values that are representative in real workloads of key-value stores [7, 16, 37]. We also evaluate the impact of different key-value sizes on the performance.

Comparisons. We compare RACE hashing with three state-of-the-art RDMA-search-friendly hashing indexes, i.e., Pilaf cuckoo hashing [33], FaRM hopscotch hashing [14], and DrTM cluster hashing [46]. Based on the optimal configurations presented in their papers, we use three-way hashing and one slot per bucket in Pilaf cuckoo hashing, four slots per main bucket, two neighborhood, and two slots per overflow bucket in FaRM hopscotch hashing, and eight slots per main or overflow bucket in DrTM cluster hashing. Moreover, since the disaggregated memory pool without CPUs cannot execute two-sided RDMA verbs, we implement these hashing indexes using only one-sided RDMA verbs as presented in Section 2.3 to facilitate a fair comparison. All key-value items are stored outside of the hash table to support variable-length keys and values. Each hash table is sized to store 100 million items.

5.2 Experimental Results and Analysis

5.2.1 Maximum Load Factor. The maximum load factor is defined as the ratio of the maximum number of stored items to the total number of slots in a hash table (including slots in main and overflow buckets), which is an important metric that affects the memory efficiency of a hash table. We insert unique keys into RACE, Pilaf cuckoo, FaRM hopscotch, DrTM cluster hash tables until an insertion failure occurs to evaluate their maximum load factors. Specifically, Pilaf cuckoo hashing reaches the maximum load factor when an insertion fails to lookup an empty slot after X cuckoo evictions. X means the maximum number of cuckoo evictions for an insertion and we evaluate maximum load factors of Pilaf cuckoo hashing under different X values. FaRM hopscotch hashing and DrTM cluster hashing reach their maximum load factors when running out of overflow buckets. As the ratio of the number of overflow buckets to that of main buckets (called overflow-to-main-bucket ratio) affects their maximum load factors, we evaluate their maximum load factors under different overflow-to-main-bucket ratios. For RACE hashing, since associativity (i.e., the number of slots per bucket) affects its maximum load factor, we evaluate its maximum load factors under different associativity.

As shown in Figure 12, the maximum load factor of RACE hashing increases with the increase of associativity. The maximum load factor of Pilaf cuckoo hashing increases with the increase of X. The maximum load factors of FaRM hopscotch hashing and DrTM cluster hashing increase with the increase of their overflow-to-main-bucket ratios.

To facilitate a fair comparison, we configure these hash tables to approach the same maximum load factor of 90% for the following experiments. RACE hashing reaches 90% when the associativity is 7. With seven slots and one header, each bucket in RACE hashing is a cache-line size, i.e., 64 B. Pilaf cuckoo hashing approaches 90% when $X = 1{,}000$. FaRM hopscotch hashing and DrTM cluster hashing approach 90% when their overflow-to-main-bucket ratios are 1/4 and 3, respectively.

5.2.2 Execution Latency. To investigate request latencies of different hashing indexes, we respectively execute 1 million search, update, deletion, and insertion requests using one thread when

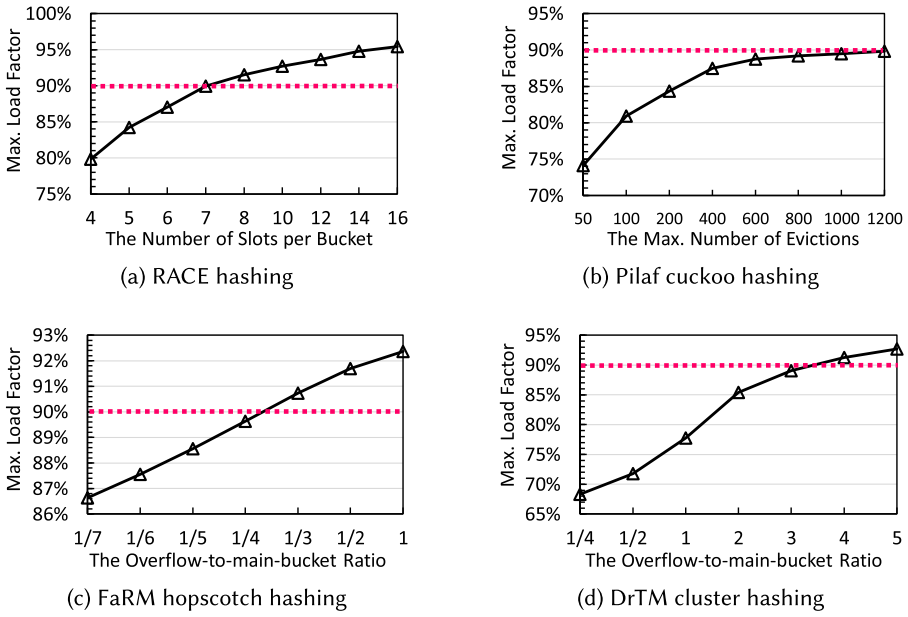

Fig. 12. Maximum load factors of different hash tables. In FaRM hopscotch hashing and DrTM cluster hashing, overflow buckets indicate the buckets linked in conflicting lists. The load factor indicates the ratio of the number of occupied slots to that of all slots in both overflow and main buckets. The overflow-to-main-bucket ratio means the ratio of the number of overflow buckets to that of main buckets.

these hash tables are in different load factors and evaluate average latencies of different requests, as shown in Figure 13. The items for search, update, and deletion are recently inserted [11].

Figure 13(a) shows the average insertion latencies of different hash tables. With the increase of load factors, the insertion latency of Pilaf cuckoo hashing exponentially increases due to causing more and more eviction operations and thus producing a large number of RDMA `WRITE`s and locks; the insertion latency of FaRM hopscotch hashing dramatically increases due to linearly probing more buckets to find empty slots and linking overflow buckets; the insertion latency of DrTM cluster hashing increases due to traversing longer bucket lists and linking new overflow buckets. The insertion latency of RACE hashing does not increase with the increase of load factors due to not causing any extra RDMA operations in which an insertion has three RTTs. When the hash tables are at the load factor of 90%, RACE hashing reduces the insertion latency by 1.9×, 8.8×, and 57.4× compared with DrTM cluster hashing, FaRM hopscotch hashing, and Pilaf cuckoo hashing, respectively.

Figure 13(b) shows the average search latencies of different hash tables and all search requests are lock-free. A search in Pilaf cuckoo hashing needs 1.6 RTTs on average to serially read buckets and 1 RTT to read key-value block. A search in FaRM hopscotch hashing needs only 2 RTTs (one reads the neighborhood and the other reads the key-value block) at a low load factor and its latency increases in a high load factor due to traversing linked buckets. The search latency of DrTM cluster hashing sharply increases with the increase of the load factor, since the bucket list becomes longer. When the hash tables are at the load factor of 90%, RACE hashing reduces the search latency by 2.3×, 1.2×, and 1.4× compared with DrTM cluster hashing, FaRM hopscotch hashing, and Pilaf cuckoo hashing, respectively.

Figure 13(c) and (d) show the average deletion and update latencies of different hash tables, which deliver similar characteristics to those of search latencies. But deletion and update latencies

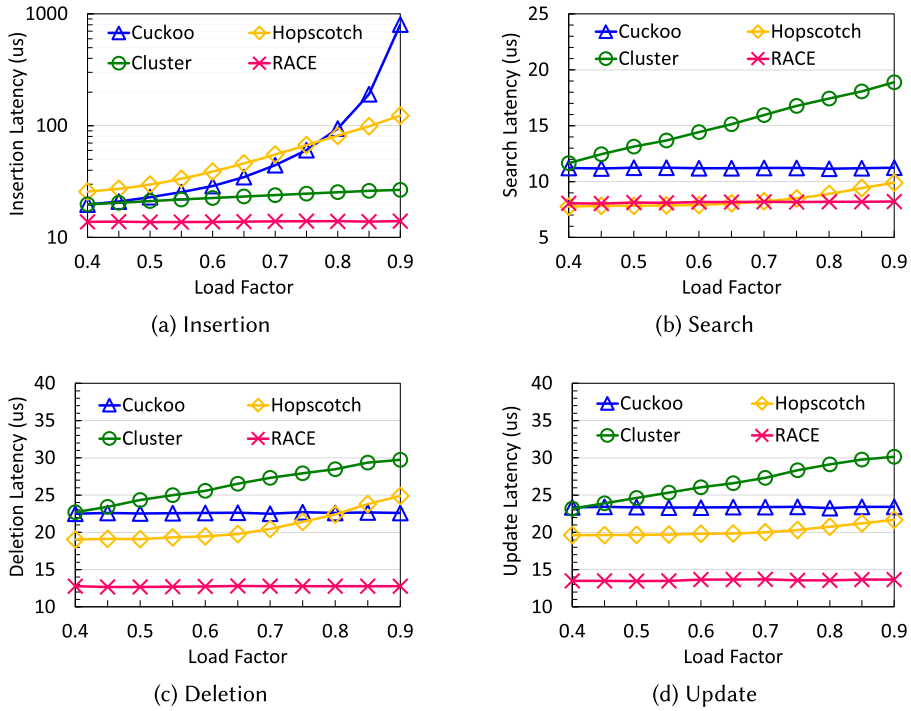

Fig. 13. Average latencies of different requests when hash tables are in different load factors.

of DrTM cluster hashing, FaRM hopscotch hashing, and Pilaf cuckoo hashing are much higher than their search latencies due to the needs of locking, unlocking, modifying slots, and unlinking buckets. The deletion and update latencies of RACE hashing are only higher than its search latency by 1-RTT latency. When the hash tables are at the load factor of 90%, RACE hashing reduces the deletion and update latencies by 1.8–2.3× and 1.6–2.2×, respectively.

5.2.3 Concurrent Throughput. To investigate the concurrent request throughput of different hashing indexes, we first load 100 million items into a hash table and then successively execute 10 million searches, updates, deletions, and insertions to evaluate the concurrent throughput of different requests. We also vary the numbers of client processes to investigate the change of the throughput with the increase of clients, as shown in Figure 14. When the number of clients is not larger than 32, they are run in one client machine. When the number of clients is 64 and 128, they are run in two and four client machines, respectively.

Figure 14(a) shows the concurrent throughput of insertion requests. The insertion throughputs of Pilaf cuckoo hashing and FaRM hopscotch hashing are much less than that of RACE hashing, of which reasons are the same as those of their high execution latencies. The insertion throughput of RACE hashing is 16.9×, 5.3×, and 1.4× on average higher than those of Pilaf cuckoo hashing, FaRM hopscotch hashing, and DrTM cluster hashing, respectively.

Figure 14(b) shows the concurrent throughput of search requests. Pilaf cuckoo hashing, FaRM hopscotch hashing, and RACE hashing have a similar search throughput that is higher than that of DrTM cluster hashing. This is because DrTM cluster hashing needs to traverse linked bucket lists. The search throughput of RACE hashing is 1.7× on average higher than that of DrTM cluster hashing.

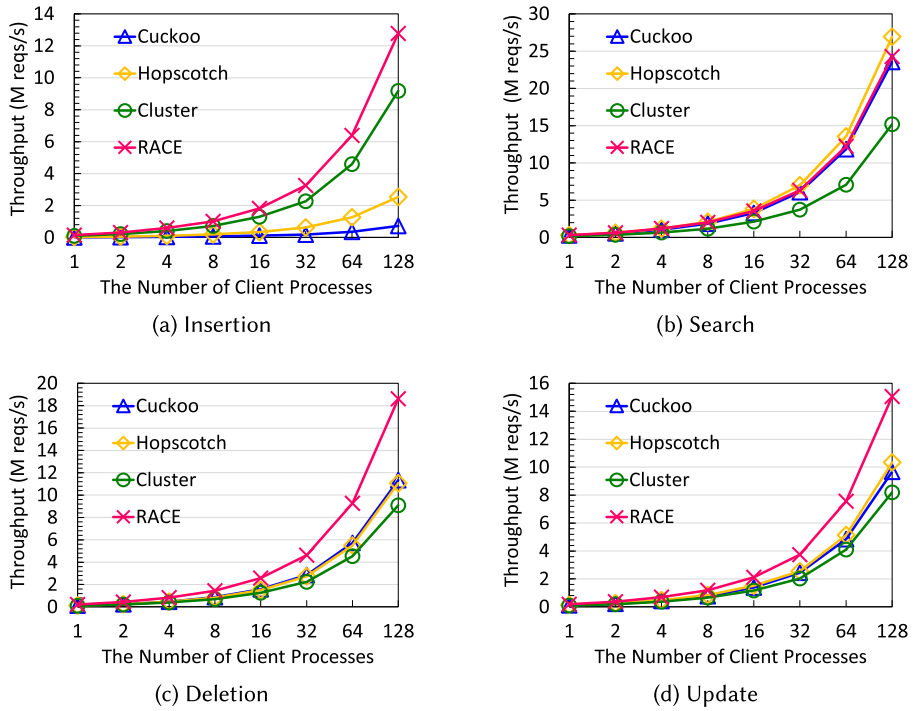

Fig. 14. Concurrent throughput of different requests when using different numbers of client processes.

Figure 14(c) and (d) show the concurrent throughput of deletion and update requests, respectively. The deletion and update throughput of RACE hashing is 1.7–2.1× and 1.5–1.9× higher than other hashing schemes due to the benefits of locking-free concurrency and RAC index structure.

5.2.4 YCSB Hybrid Workloads. To evaluate the throughput of different hashing indexes under YCSB hybrid workloads, we first load 90 million items into a hash table and then respectively run hybrid search/insertion workloads with different ratios. All tests use 128 client processes. The experimental results are shown in Figure 15. We observe that the throughput of all hashing indexes increases with the increase of search/insertion ratios, and RACE hashing performs the best for all search/insertion ratios due to having both high search and insertion performance. Compared with DrTM cluster hashing, FaRM hopscotch hashing, and Pilaf cuckoo hashing, RACE hashing improves the performance of hybrid workloads by 1.4×, 4.9×, and 13.7×, respectively, when the search/insertion ratio is 10%/90%.

5.2.5 Variable-length Values. We increase the size of the **key-value (KV)** block from 64 B to 8 KB to evaluate the impact of variable-length KV sizes on the performance of RACE hashing, as shown in Figure 16. With the increase of the KV size, the latencies of insertion, deletion, update, and search requests increase due to reading and writing larger data. When the KV size is no larger than 512 B, the increase of latencies is slight.

5.2.6 Extendible Remote Resizing. To support extendible remote resizing, we propose two techniques, i.e., the SRCD cache and concurrent access during resizing as presented in Section 3.4. We investigate the impact of the two techniques on the performance of RACE hashing.

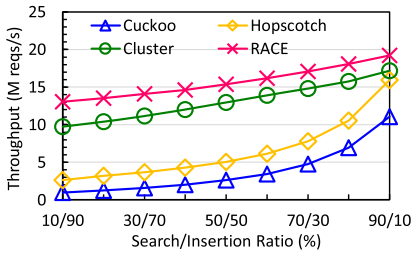

Fig. 15. Hybrid workloads.

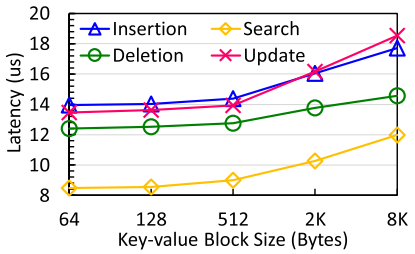

Fig. 16. Variable KV sizes.

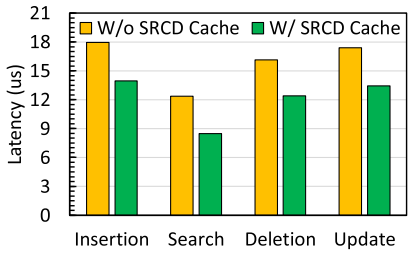

Fig. 17. The SRCD cache.

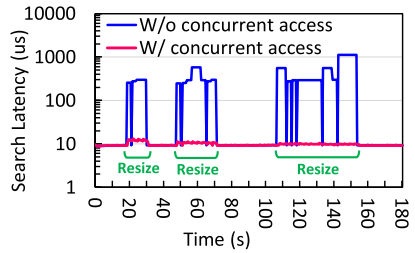

Fig. 18. Concurrent access.

Figure 17 shows the performance of RACE hashing with and without the SRCD cache. We observe that using the SRCD cache reduces 23%, 32%, 24%, and 23% of insertion, search, deletion, and update latencies, respectively. This is because using the SRCD cache reduces one extra RDMA READ for accessing the directory.

To investigate concurrent access during resizing, we run two clients of which one executes insertions to trigger multiple resizings (the GD increases from 2 to 5) and the other executes random searches at the same time. We evaluate the average search latencies of RACE hashing with and without concurrent access during resizing as shown in Figure 18. Without the concurrent access, the average search latency during the resizing significantly increases, since the searches stall until a resizing is completed. With the concurrent access, the average search latency during the resizing does not significantly increase and thus is about two orders lower than that without concurrent access.

5.2.7 Associativity. In the previous experiments, we fix the associativity of RACE hashing to 7 to faciliate a fair comparison with other hash tables. To investigate the impact of associativity on request latencies of RACE hashing, we execute 1 million search, update, deletion, and insertion requests, respectively, on RACE hash tables with different associativities using one thread. The initial load factors of all hash tables are set to 0.5. Figure 19 shows the request latencies of four RACE hashing operations under different associativities. When we increase the associativity from 4 to 32, the latencies of insertion, search, deletion, and update increase by 6.8%, 0.8%, 1.5%, and 4.7%, respectively, which indicates that associativity has little impact on the latencies of RACE hashing operations.

To investigate the impact of associativity on the concurrent request throughput of RACE hashing, we first load 100 million items into RACE hash tables of different associativities and then execute 10 million searches, updates, deletions, and insertions using 64 clients. To facilitate a fair comparison, we build RACE hash tables with different associativities with the same amount of memory. As shown in Figure 20, the throughputs of all operations decrease significantly with the

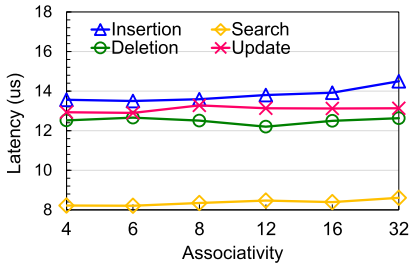

Fig. 19. Request latencies of different operations under different associativities.

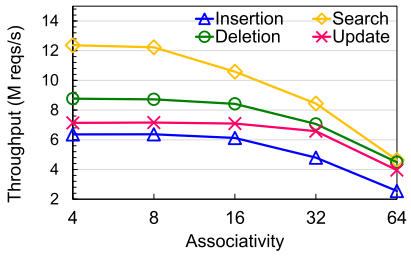

Fig. 20. Concurrent throughputs of different operations under different associativities.

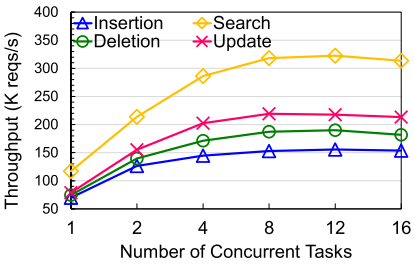

Fig. 21. Client-side throughputs with different numbers of concurrent co-routine tasks.

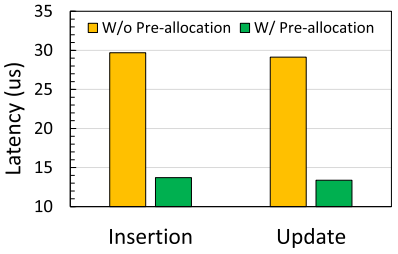

Fig. 22. Latencies with pre-allocation.

increase of associativity. When we increase the associativity from 4 to 64, the throughputs of insertion, search, deletion, and update decrease by 59%, 62%, 48%, and 44%, respectively. The reason is that large associativity causes read amplification and the network bandwith becomes the bottleneck. Nevertheless, increasing associativity improves the maximum load factor of RACE hashing as shown in Figure 12(a). Therefore, setting the associativity is a tradeoff between the performance and memory utilization of the RACE hash table.

5.2.8 Client-side Request Handling. As presented in Section 4.1, we use co-routines to implement all the operations of RACE hashing to improve the throughput of client-side request handling. To demonstrate the effect of co-routine, we first load 1 million key-value pairs into a RACE hash table and then execute 1 million search, update, deletion, and insertion requests, respectively, using one client thread with different numbers of concurrent co-routine tasks. When the number of concurrent tasks is set to 1, the client thread executes all requests sequentially.

Figure 21 shows the throughputs of handling client-side insertion, search, deletion, and update requests with different numbers of concurrent co-routine tasks. When we increase the number of concurrent co-routine tasks from 1 to 16, the throughputs of insertion, search, deletion, and update are improved by 2.2×, 2.6×, 2.7×, and 2.4×, respectively. We also observe that the increase in the throughputs slows down with the increase in the number of concurrent tasks and the throughputs of all operations stop increasing when the number of tasks is larger than 8. This is because with many concurrent tasks, the client-side CPU is exhausted and becomes the new bottleneck.

5.2.9 Memory Allocation. To investigate the benefit of pre-allocation machanism proposed in Section 4.2, we execute 1 million insertion and update operations, respectively, on the RACE hash tables with and without the pre-allocation machanism. When pre-allocation is disabled, each

insertion or update launches one RPC to the memory pool to allocate one key-value block. We initialize the load factors of two RACE hash tables to 0.5. As shown in Figure 22, the latencies of insertion and update with pre-allocation are more than 2× lower than the latencies of insertion and update without pre-allocation. The reason is that the overheads of memory allocations are removed from the critical paths of insertion and update operations when the pre-allocation machanism is enabled.

6 DISCUSSION

Concurrency Correctness. RACE hashing follows the concurrency correctness condition of *no lost keys* [32]: "a $get(K)$ operation must return a correct value for K, regardless of concurrent writers." Specifically, when a search and an update run concurrently, the search can return either the new or the old value, while both of them should be unbroken and atomic. When a search and a deletion run concurrently, the search can return no key or the value that will be deleted.

Resizing Execution. In our current implementation, the client triggering the resizing itself performs the resizing. To improve the implementation, the client can create a background client/thread to perform the resizing.

Hardware Failure. Handling hardware failures including network failure, memory failure, and client CPU failure in the disaggregated memory architecture is complicated and tough. For example, after locking a directory entry, the client fails. To handle this failure, we need to enable other clients to perceive the failed client and release the lock directory entry or use the lease-based lock [46]. Our article mainly focuses on the design of hashing index for disaggregated memory and plan to extend RACE hashing to support the handle of hardware failures in the future work.

7 RELATED WORK

Memory Disaggregation. Memory disaggregation has recently received widespread attentions due to providing significant benefits for datacenters on resource utilization and scaling. Existing work studies various components in datacenters to support memory disaggregation including operating systems [40], hardware architectures [30, 31], memory managements [4, 39, 42–44], networks [1, 9, 12, 41], and new requirements [5, 19]. RACE hashing focuses on the design of index structures in the disaggregated memory that is orthogonal to these works.

Hashing Indexes on RDMA. With the wide use of RDMA in modern datacenters, RDMA-search-friendly hashing indexes have been intensively studied [14, 33, 46]. These hashing indexes are designed for traditional monolithic servers, which however fails to efficiently work on the new disaggregated memory architecture, due to producing a large number of RDMA operations when executing IDU requests. RACE hashing is the first hashing index designed for disaggregated memory, in which all index requests can be efficiently executed by using only one-sided RDMA operations. Moreover, KV-direct [26] leverages programmable NICs with FPGA to offload hashing index operations, which is orthogonal to our article that does not reply on FPGA.

Concurrent Hashing Indexes. Different concurrent hashing indexes are proposed to deliver high access throughput. MemC3 [16] uses a global lock to multi-reader and single-writer concurrency for concurrent cuckoo hashing. Libcukoo [28] leverages fine-grained locking to achieve multi-reader and multi-writer concurrent cuckoo hashing. Existing work [10, 35, 48] also proposes concurrent hashing indexes for persistent memory. However, all these existing schemes focus on concurrent access to local memory. Unlike them, our RACE hashing addresses the challenge of concurrent access to remote memory in hash indexes and enables all index requests to be executed in a lock-free manner.

8 CONCLUSION

This article proposes RACE hashing, a one-sided RDMA-conscious extendible hashing index for disaggregated memory with lock-free remote concurrency control and efficient remote resizing. The hash table structure is designed to be one-sided RDMA-conscious, achieving that all index requests can be executed using only one-sided RDMA verbs while delivering high performance with constant-scale worst-case time complexity. Moreover, RACE hashing leverages a lock-free remote concurrency control scheme to enable index requests to be concurrently executed in a lock-free manner, and a stale-read client directory cache scheme to reduce one extra RDMA read for accessing the directory while guaranteeing the correctness of stale cache reads. We also achieve concurrent access to the subtable that is being resized. Experimental results show that RACE hashing outperforms state-of-the-art distributed in-memory hashing indexes by up to 13.7× in YCSB hybrid workloads.

REFERENCES

[1] 2022. Gen-Z Technology. Retrieved from https://genzconsortium.org/.

[2] 2022. Memcached—A Distributed Memory Object Caching System. Retrieved from https://memcached.org/.

[3] 2022. Redis. Retrieved from https://redis.io/.

[4] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: A simple abstraction for remote memory. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'18)*. 775–787.

[5] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'19)*. 120–126.

[6] Krste Asanović and David Patterson. 2014. FireBox: A hardware building block for 2020 warehouse-scale computers. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'14)*.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*. 53–64.

[8] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D. Underwood, and Robert C. Zak. 2015. Intel Omni-path architecture: Enabling scalable, high performance fabrics. In *Proceedings of the IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI'15)*. IEEE, 1–9.

[9] Amanda Carbonari and Ivan Beschasnikh. 2017. Tolerating faults in disaggregated datacenters. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets'17)*. 164–170.

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

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. 143–154.

[12] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. 2015. R2C2: A network stack for rack-scale computers. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*. 551–564.

[13] Ana Lúcia De Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2, Article 6 (February 2009), 31 pages.

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

[15] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H Raymond Strong. 1979. Extendible hashing-a fast access method for dynamic files. *ACM Trans. Database Syst.* 4, 3 (1979), 315–344.

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

[17] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. 2015. Beyond processor-centric operating systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS'15)*.

[18] Michael J. Franklin, Michael J. Carey, and Miron Livny. 1997. Transactional client-server cache consistency: Alternatives and performance. *ACM Trans. Database Syst.* 22, 3 (1997), 315–363.

[19] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 249–264.

[20] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: Coroutine-oriented main-memory database engine. arXiv:2010.15981. Retrieved from https://arxiv.org/abs/2010.15981.

[21] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch hashing. In *Proceedings of the International Symposium on Distributed Computing (DISC'08)*. Springer, 350–364.

[22] Hewlett Packard Corporation. 2015. The Machine: A New Kind of Computer. Retrieved from https://www.hpl.hp.com/research/systems-research/themachine/.

[23] Intel Corporation. 2022. Intel Rack Scale Design Architecture. Retrieved from https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[24] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design guidelines for high performance RDMA systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'16)*. 437–450.

[25] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 468–479.

[26] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. 137–152.

[27] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. 2015. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. 476–488.

[28] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*. 1–14.

[29] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*. 21–35.

[30] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. 267–278.

[31] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level implications of disaggregated memory. In *Proceedings of the IEEE International Symposium on High-Performance Comp Architecture (HPCA'12)*. IEEE, 1–12.

[32] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. 183–196.

[33] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15)*. 103–114.

[34] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.* 12, 10 (2001), 1094–1104.

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

[36] Nhan Nguyen and Philippas Tsigas. 2014. Lock-free cuckoo hashing. In *Proceedings of the IEEE 34th International Conference on Distributed Computing Systems (ICDCS'14)*. IEEE, 627–636.

[37] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*. 385–398.

[38] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algor.* 51, 2 (2004), 122–144.

[39] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-performance, application-integrated far memory. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 315–332.

[40] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 69–87.

[41] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A network architecture for disaggregated racks. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. 255–270.

[42] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'20)*. 33–48.

[43] Shin-Yeh Tsai and Yiying Zhang. 2017. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. 306–324.

[44] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A memory-disaggregated managed runtime. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 261–280.

[45] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy lock-free indexing in non-volatile memory. In *Proceedings of the IEEE 34th International Conference on Data Engineering (ICDE'18)*. IEEE, 461–472.

[46] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. 87–104.

[47] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the abyss: An evaluation of concurrency control with one thousand Cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220.

[48] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 461–476.

Received November 2021; revised November 2021; accepted January 2022