DHash: Dynamic Hash Tables With Non-Blocking Regular Operations

Junchang Wang, Dunwei Liu, Xiong Fu, Fu Xiao, and Chen Tian

Abstract—Once started, existing hash tables cannot change their pre-defined hash functions, even if the incoming data cannot be evenly distributed to the hash table buckets. In this paper, we present DHash, a type of hash table for shared memory systems, that can change its hash function and rebuild the hash table on the fly, without noticeably degrading its service. The major technical novelty of DHash stems from an efficient distributing mechanism that can atomically distribute every node when rebuilding, without locking the corresponding hash table buckets. This not only enables non-blocking lookup, insert, and delete operations, but more importantly, makes DHash independent of the implementation of hash table buckets, such that DHash allows programmers to select the set algorithms that meet their requirements best from a variety of existing lock-free and wait-free set algorithms. Evaluations show that DHash can efficiently change its hash function on the fly. Moreover, when rebuilding, DHash consistently outperforms the state-of-the-art hash tables in terms of throughput and response time of concurrent operations, at different concurrency levels, and with different operation mixes and average load factors.

Index Terms—Concurrent programming, parallelism and concurrency, dynamic hash tables

1 Introduction

Typically, a hash table consists of an array of *buckets*. Nodes to be inserted into the hash table are first reduced to 32-bit or 64-bit *hash value* by using a pre-defined *hash function*. The hash table takes the hash value modulo the size of its bucket array to determine the buckets where the corresponding nodes will be stored. Once two or more distinct nodes are mapped to the same bucket, a *hash collision* has happened. One typical approach to dealing with hash collisions is *separate chaining*, in which each hash bucket contains a list of nodes mapped to the bucket. The average length of the lists, referred to as *average load factor*, is defined as the total number of nodes in the hash table divided by the size of the bucket array. In general cases, given a specified average load factor, hash tables offer the advantage of constant-time lookup operations, such that they have been widely used throughout computer systems.

Most existing hash tables, however, lacks robustness; the pre-defined hash functions cannot be changed even if they cannot evenly distribute incoming data to the hash table buckets. For example, an adversary can launch algorithmic complexity attacks [1] by first spying out the details of the hash function and then creating malicious nodes that will be hashed to a few buckets. This causes a few buckets (henceforth referred to as *target buckets*) to contain many more nodes than the average load factor, effectively making the hash table unavailable for legitimate requests accessing these buckets. This robustness issue has affected the hash table implementations in a long list of operating systems and programming languages, including the Linux kernel [2], Perl [1], PHP [3], and .NET [4].

Researchers have proposed various techniques to address this robustness issue. The most well-studied solution is *universal hashing* [5], which, by randomly selecting a hash function from a set of well-designed hash functions, can, in theory, provide good average-case hashing performance and address the above robustness issue. Unfortunately, empirical evaluation shows that *universal hashing* is not fit for applications such as Perl [6], which is heavily used to process strings that are often selected from restricted character sets, and thus have an unexpected distribution property. Moreover, emerging research shows that it is possible for the adversary to observe or guess the random choice of the pre-defined hash function, rendering universal hashing useless [1], [7], [8], [9].

In this paper, we first demonstrate that rebuilding a hash table by dynamically changing its hash function on the fly is a promising approach to build robust hash tables. Specifically, once the pre-defined hash function cannot evenly distribute incoming data to the buckets, a new instance of the hash table with a distinct, well-designed hash function is created. Then, the existing nodes in the old hash table are distributed to the new one. During the time period of distributing nodes, both hash tables are used to serve concurrent *insert*, *delete*, and *lookup* operations (henceforth simply *regular* operations). After all nodes have been distributed to the new hash table, the old hash table can be deleted. By changing the hash function, the new hash table can distribute existing nodes evenly over all buckets, and, therefore, provide the

- *Junchang Wang, Dunwei Liu, Xiong Fu, and Fu Xiao are with the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China. E-mail: {wangjc, 1020041113, fux, xiaof}@njupt.edu.cn.*
- *Chen Tian is with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210008, China. E-mail: tianchen@nju.edu.cn.*

Manuscript received 24 Sept. 2020; revised 5 Jan. 2022; accepted 8 Feb. 2022. Date of publication 14 Feb. 2022; date of current version 2 June 2022.
This work was supported in part by the National Science Fund for Distinguished Young Scholars of China under Grant 62125203, and in part by the Key Program of the National Natural Science Foundation of China under Grants 61932013 and 62072228.
(Corresponding author: Fu Xiao.)
Recommended for acceptance by R. Vaidyanathan.
Digital Object Identifier no. 10.1109/TPDS.2022.3151499

expected constant-time lookup operations. We use the term *dynamic* to describe a hash table algorithm that can provide this flexibility feature, and refer to the function that dynamically changes its hash function as *rebuild*. Researchers have proposed hash tables that can only enlarge or shrink their bucket sizes by constant factors but cannot change hash functions [10], [11], [12], [13], which we refer to as *resizable* hash tables. In Section 6.3, we demonstrate that for each run, resizable hash tables can only cut the list length of the target buckets by half, and hence cannot resolve the robustness issue. We thus mainly focus on dynamic hash tables. Note that in this paper, for ease of presentation, we only say that a dynamic hash table can change its hash function, even though it actually can change its hash function and/or hash seed. In contrast, a resizable hash table cannot change any of them, and the only aspect a resizable hash table can change is the size of its bucket array. Moreover, in some sense, a *dynamic* hash table is the hash table that is *resizable* and can *rebuild* by using a different function.

The core problem in designing a dynamic hash table is how to atomically and efficiently distribute every node from the old hash table to the new one. This poses the following unique challenge: Even though the delete and insert operations on a single hash table can be atomic and non-blocking [14], [15], [16], [17], there are no non-blocking approaches that can atomically distribute every node from the old hash table to the new one. Consequently, prior dynamic hash tables [18], [19] all depend on locking mechanisms and need to first lock the corresponding buckets in the old and the new hash tables, before distributing each node (see Related Work for the pseudo-code). However, it is well known that locking can prevent regular operations from executing simultaneously, leading to unexpected delays or even suspension in a busy system. In Section 6.2, we demonstrate that for a typical hash-table usage pattern, when a rebuild operation is in progress, the 99.9%-percentile response time of these lock-based dynamic hash tables' lookup operations reaches 5 microseconds, 3.6 times worse than the normal case, adding significant delays to normal operations from legitimate users.

This paper presents DHASH, a novel non-blocking Dynamic Hash table. The major technical novelty of DHASH stems from a new *distributing mechanism* that can atomically and efficiently distribute each node without needing to acquire any per-bucket mutex locks. (To serialize concurrent rebuilding attempts, the rebuild operation uses a global lock, which, however, can never block any regular operations. We discuss this in Section 4.3.) Experimental results show that the distributing mechanism is lightweight and does not noticeably degrade the performance of concurrent regular operations. Furthermore, the distributing mechanism is independent of the implementation of hash table buckets, such that DHASH can utilize a variety of existing lock-free and wait-free set algorithms as the implementation of its buckets. This modular design allows programmers to make trade-offs among their own DHASHs' progress guarantee, performance, and engineering efforts, and to select the set algorithms that meet their requirements best, bridging the gap between building robust hash tables and utilizing existing non-blocking set algorithms from the parallel-processing community.

We prove the correctness of our key techniques and implement two versions of DHASH that respectively provide lock-free and wait-free lookup operations. Experimental results show that DHASH can effectively change its hash function and seed. Moreover, our distributing mechanism is lightweight; when rebuilding, DHASH consistently outperforms the alternatives in terms of throughput and response time of concurrent regular operations, at different concurrency levels, and with different operation mixes and average load factors. Furthermore, we demonstrate that DHASH's rebuild operation is scalable; the parallelized rebuild operation of DHASH with eight or more rebuilding threads outperforms the state-of-the-art alternatives.

The rest of the paper is organized as follows. We first discuss related work in Section 2. Section 3 gives an overview of DHASH and the distributing mechanism. Section 4 presents the details of the algorithm. We prove the correctness of DHASH in Section 5, present evaluations in Section 6, and conclude in Section 7.

2 Related Work

This section sketches a high-level overview of the existing dynamic and resizable hash tables.

Dynamic Hash Tables. Xu *et al.* designed a dynamic hash table [18] for the management of IGMP packets in the Linux kernel in 2010. The key idea behind Xu's algorithm is to manage two sets of pointers in each node, so that regular operations traverse one set of pointers while the rebuild operation is updating the other set. The two sets are exchanged upon the completion of every rebuild operation. Xu's algorithm is straightforward and easy to be implemented, but it has two major drawbacks in practice. (1) Locking mechanisms are used to serialize concurrent update and rebuild operations. (2) A linked list algorithm must be customized by adding an extra set of pointers before it can be used by Xu's hash table. In contrast, DHASH overcomes these drawbacks in its design.

Based on Triplett *et al.*'s ATC'11 paper [11], Graf introduced a dynamic hash table into the Linux kernel in 2014 [19], and this algorithm has been widely used in the kernel. Graf's hash table maintains a single pointer in each node, and utilizes per-bucket mutex locks to synchronize concurrent update and rebuild operations on each bucket. The skeleton of the rebuild operation of Graf's dynamic hash table is as follows.

```
for each bucket htbp in the old hash table
  mutex_lock(htbp);
  for each node htnp in htbp
    remove htnp from htbp;
    htbp_new = new_hash(htnp);
     mutex_lock(htbp_new);
    insert htnp into htbp_new;
     mutex_unlock(htbp_new);
  mutex_unlock(htbp);
```

Correspondingly, the insert and delete operations of Graf's dynamic hash table must first acquire the corresponding per-bucket mutex lock before they can operate on a bucket. Graf's algorithm is a practical design. However, this algorithm has the following drawbacks. (1) It uses locks to serialize updates to the same bucket. (2) It maintains unordered lists as its buckets and temporarily concatenates two lists when rebuilding.

Perl (versions between 5.8.2 and 5.16.2) uses a much more coarse-grained locking mechanism when rebuilding; the rebuild operation needs to acquire a global lock before distributing every node, preventing concurrent insert and delete operations from making any progress, even though they work on different buckets.

Lock-Free Resizable Hash Tables. The resizable hash tables [10], [11], [13] do not change their hash functions; all they can do is enlarge or shrink their bucket sizes by a constant factor. One classic resizable hash table [10] was presented by Ori Shalev and Nir Shavit in 2006. To atomically distribute every node from the old hash table to the new one, Shalev and Shavit's algorithm introduces a novel data structure called "recursive split-ordered list" which is fundamentally a lock-free linked list. Each hash table maintains a single "recursive split-ordered list," and all nodes in the hash table are chained in this list. This hash table gradually assigns the bucket pointers to the places in the list where a sublist containing the requested node can be found. In solving the atomic-distribution problem when resizing, Shalev and Shavit's algorithm does not move nodes among the buckets. Instead, it moves the buckets among the nodes by referencing buckets to the proper nodes in the list.

Even though the split-ordered list is lock-free, it has drawbacks in practice. (1) The algorithm must first bit-reverse the hash results of the keys before performing any regular operations. Unfortunately, the bit-reverse operation is inefficient on X86 and Power9, where hardware does not provide native instructions. (2) Evaluations show that when the pre-defined hash function cannot evenly distribute incoming data, resizing cannot effectively help a hash table recover.

Open-Addressing Based Hash Tables. Researchers have proposed several resizable hash tables [20], [21] based on *open addressing* [22]. In particular, Maier *et al.* presented a resizable hash table [21] based on *linear probing* [22]—a form of open addressing— demonstrating that it is possible to build fast, open-addressing based resizable hash tables. The main technical contributions of Maier *et al.*'s algorithm and that of DHash are orthogonal. We focus on separate-chaining based hash tables in this paper and leave applying our distributing mechanism to open-addressing based hash tables (e.g., Maier *et al.*'s algorithm) as future work.

3 DHash Algorithm Overview

This section first presents the system model and key challenge in designing dynamic hash tables, and then sketches a high-level overview of the distributing mechanism of DHash, leaving technical details to Section 4.

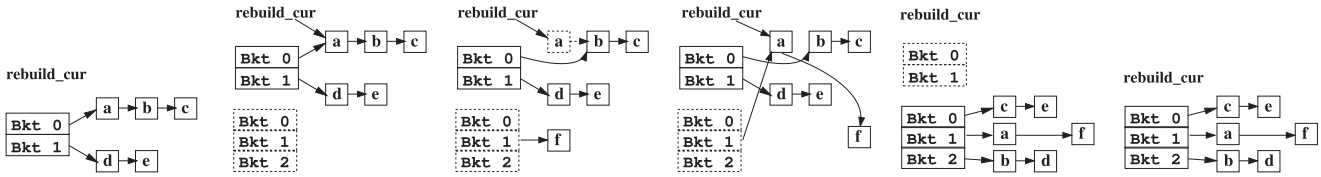
We design DHash for an *asynchronous shared memory system* [23] with multiprocessors. On such a system, a program is executed by p deterministic threads, where p may exceed the number of physical processors. A scheduler decides which threads to run and may suspend the execution of any thread at any time for arbitrarily long. We assume a *TSO* memory model [24] in this paper.

The key challenge in designing a dynamic hash table is to atomically and efficiently distribute every node from the old hash table to the new one. To distribute each node, the rebuild operation must update two hash tables. Even though the delete and insert operations on a single hash table can be atomic and non-blocking [14], [15], [16], [17], there are no non-blocking approaches that can atomically move a node from the old hash table to the new one. Prior research addressed this challenge by (1) maintaining two sets of linked lists in each bucket [18], (2) synchronizing the rebuild and other regular operations by using locks [18], [19], (3) maintaining unordered linked lists [11], [19], and/or (4) degenerating dynamic hash tables to resizable hash tables [10]. These approaches, however, sacrifice the algorithms' generality and/or performance.

Distributing Mechanism. This paper presents a novel *distributing mechanism* that can atomically and efficiently distribute every node from the old hash table to the new one, without acquiring any mutex locks. The basic idea behind our distributing mechanism is that we can synchronize the rebuild operation and concurrent regular operations by managing them to access hash tables and shared variables in specified orders, rather than serializing them by using expensive synchronization mechanisms such as locking. Specifically, to distribute a node, the rebuild operation of DHash first deletes the node from the old hash table and then inserts it into the new hash table, without acquiring any mutex locks. The process of distributing the node leads to a short time period during which in neither the hash tables can this node be found. We call this a node's *hazard period*. To allow other concurrent regular operations to be able to access the node, DHash employs a global pointer that always points to the node that is in the *hazard period*. Consequently, when a rebuild operation is in progress, concurrent regular operations need to check different locations because a node may reside in either the new or the old hash table, or is referenced by the global pointer. In Section 4, we prove that if regular operations search both hash tables and check the node currently in the *hazard period* in the specified orders, they can always find the node with the matching key and successfully apply the specified operations on it.

Example of Rebuilding. To illustrate how DHash's distributing mechanism works, we use the example hash table shown in Fig. 1. In this hash table, DHash consists of two buckets: *Bkt 0* and *Bkt 1*. *Bkt 0* contains three nodes (`a`, `b`, and `c`), and *Bkt 1* contains two nodes (`d` and `e`). We assume that the new hash table contains three buckets, and that its user-provided hash function maps all of the keys to the new three-buckets array. The rebuild operation then performs a hash table traversal and distributes every node in the old hash table to the new one. Fig. 1 illustrates the process of distributing the node `a`, with the initial state shown in Fig. 1a and with the time advancing from figure to figure.

Specifically, the node `a` is first pointed to by the global pointer *rebuild_cur*, resulting in the state shown in Fig. 1b. Then, `a` is removed from the old hash table and enters its *hazard period*, shown in Fig. 1c. When `a` is in the *hazard period*, other concurrent regular operations can access it via the global pointer *rebuild_cur*. Without loss of generality, we assume that when the rebuild operation is in progress, other regular operations concurrently insert a new node `f` into the new hash table, shown in Fig. 1c. Then, the node `a` is inserted into the new hash table, shown in Fig. 1d. After it has been successfully inserted into the new hash table, *rebuild_cur* is set to NULL. The rebuild operation traverses the old hash table and distributes every node to the new



(a) Initial state. The hash table contains two buckets. (b) Rebuild operation starts. *rebuild_cur* first points to node a. (c) Node a is deleted from the old hash table, and is in its *hazard period*. (d) Node a is inserted in the new table. *rebuild_cur* is then set to *NULL*. (e) Rebuild traverses the old hash table, and distributes all of the nodes. (f) Rebuild waits for prior operations, then frees the old hash table.

Fig. 1. The workflow of DHASH's rebuild operation. *Bkt* is short for *Bucket*. *rebuild_cur* points to the node in *hazard period*.

hash table, shown in Fig. 1e. After that, the rebuild operation waits for all prior unfinished operations to complete before safely reclaiming the old hash table, shown in Fig. 1f.

Fig. 1 illustrates the following key reasons that DHASH consistently outperforms prior dynamic and resizable hash tables. (1) By changing the hash function, DHASH effectively cuts the list length of the target buckets that contain many more nodes than the average load factor. Therefore, concurrent regular operations can avoid traversing the lists that are unacceptably long. (2) Allowing other threads to access the node that is in the *hazard period* is the key reason that distributing a node is not necessary to involve expensive locking mechanisms in distributing the node. (3) In sharp contrast to prior algorithms [11], [19] that can only distribute the tail nodes of unordered lists, DHASH utilizes efficient set algorithms (e.g., lock-free skip lists [16]) and always distributes the head nodes. We demonstrate these advantages in Section 6.

4 DHASH IMPLEMENTATION

The design of DHASH presented in Section 3 leads to a relatively straightforward implementation, which is the subject of this section. Section 4.1 introduces concepts that provide a foundation for our design. We then discuss in detail how DHASH synchronizes rebuild operations and concurrent regular operations.

Note that even though we use the RCU mechanism in this paper, it can be replaced by other synchronization mechanisms such as reference counters [15] and hazard pointers [25]. There is no direct impact on the distributing mechanism in DHASH by using the alternative techniques. However, they can lead to higher performance penalties. While using reference counters and hazard pointers, additional memory fence instructions are required each time a lookup operation moves forward to a new node. Updating the counter within each node and referencing a hazard pointer to the node may cause cache misses. Moreover, by using RCU, the rebuild operation of DHASH takes the benefit of the "waiting" mechanism provided by RCU [26] (in particular, the *synchronize_rcu()* and *call_rcu()* primitives). In contrast, while using reference counters or hazard pointers, programmers need to implement the "waiting" mechanism by hand, which requires much more engineering effort.

4.1 Preliminaries

Read-Copy Update. RCU was originally developed for operating systems such as IBM's K42 [27], and nowadays has been widely used in the Linux kernel [28] and user-space applications [29]. RCU works as a synchronization mechanism to address read-write conflicts. In particular, it distinguishes between the read-side code and the write-side code and has the following primitives:

- *rcu_read_lock()* / *rcu_read_unlock()*defines the read-side critical section. Each time a thread wants to access shared variables, it accesses them in a read-side critical section, which begins with the primitive *rcu_read_lock()*and ends with the primitive *rcu_read_unlock()*. Within a read-side critical section, the thread is safe to access the shared resources; it does not need to worry if these resources could be freed simultaneously by other threads.

  The hash table regular operations of DHASH are protected by RCU read-side critical sections. That is, before invoking the function *ht_lookup()*, *ht_insert()*, or *ht_delete()*, which will be discussed later, a caller must have entered an RCU read-side critical section by invoking *rcu_read_lock()*.
- *synchronize_rcu()* works as a wait-for-readers barrier. This function blocks until all pre-existing RCU read-side critical sections have completed. It can be used in the rebuild operation of DHASH, for example, to ensure that any lookup operations that might have references to the old hash table complete before freeing the old hash table.
- *call_rcu()* is an asynchronous version of *synchronize_rcu()*. Instead of blocking, *call_rcu()* register a callback function and argument, which can be invoked by a separate thread after all ongoing RCU read-side critical sections have completed. The caller to *call_rcu()* can thus continue without blocking. The function *call_rcu()* is particularly useful in situations where it is illegal to block or where update-side performance is critically important. For example, *call_rcu()* is used in DHASH to safely reclaim nodes memory without blocking the delete operation.

RCU synchronizes readers with writers by using constrained access order, instead of shared variables [30]. Any RCU-protected node accessed by a reader is guaranteed to remain unreclaimed until the reader completes its access and calls *rcu_read_unlock()*. The production-quality implementations of *rcu_read_lock()* and *rcu_read_unlock()* are extremely lightweight; they have exactly zero overhead in the Linux kernels built for production use with *CONFIG_PREEMPT=n* [28] and have extremely close to zero overhead in user-space applications when the *QSBR flavor* model is used [29], such that readers of RCU-based data structures can execute as fast as single-threaded programs.

API of DHash *'s Bucket*. DHash is modular, and its buckets can be implemented by using any lock-free or wait-free set algorithm that implements the Application Programming Interface (API) shown in Algorithm 1.

Algorithm 1: Structures and API of DHash's Bucket.

```
1: struct node {long key; <node *ptr, flag> next};
   /* Has been logically removed by delete operation. */
2: define LOGIC_RM (1UL << 0)
   /* Is being distributed by rebuild operation.  */
3: define IN_HAZARD (1UL << 1)
4: struct list {node *head};
5: struct snapshot {node **prev, *cur, *next};
   /* Search the node in list htbp .                 */
6: list_find(list *htbp, key, snapshort *sp)
   /* Insert node htnp into list htbp .              */
7: list_insert(list *htbp, node *htnp)
   /* Search the node in list htbp , set the node's
   flag , and try to physically delete the node.    */
8: list_delete(list *htbp, long key, long flag)
```

The data structures and API of DHash's bucket are shown in Algorithm 1. The structure *list*, which will be used as the implementation of DHash's buckets, is fundamentally a chain of *nodes*. For each *node*, the *key* field holds the key value, and the *next* field, which is a pointer, consists of two parts called *ptr* and *flag*. The *ptr* field points to the following node in the linked list if any, or has a *NULL* value otherwise. Since pointers are at least word aligned on all currently available architectures, the two least significant bits of *next* are used as the *flag* field indicating if the node is in a special state. The least significant bit, denoted as *LOGIC_RM*, indicates that a node has been logically removed by a delete operation. The second least significant bit, denoted as *IN_HAZARD*, indicates that a rebuild operation is distributing this node from the old hash table to the new one, and hence this node is in its *hazard period*. The difference between these two states is whether the node memory will be reclaimed when the node is physically removed from the list, which we will discuss in detail in the following paragraphs.

To return the search result of function *list_find()* to the function invoking it, we borrowed the definition of the structure *snapshot* from Michael's classic lock-free linked list [14]. Specifically, each time we want to search a node, an instance of *snapshot* is passed to *list_find()*. Upon the completion of *list_find()*, it is guaranteed that the *cur* field of the snapshot points to the list node containing the value that is greater than or equal to the specified search key.

The set algorithm is required to implement three functions: *list_find()*, *list_insert()*, and *list_delete()*. The function *list_delete ()* takes the third parameter *flag*. If *flag* is set to *LOGIC_RM*, *list_delete()* deletes the matching node from the list and reclaims the node memory. In contrast, if *flag* is set to *IN_HAZARD*, the node memory will not be reclaimed because the node will be inserted into the new hash table. The function *list_delete()* does not block because it uses *call_rcu()* to asynchronously reclaim a node. Note that *call_rcu()* is safe to be invoked within an RCU read-side critical section [30].

Choice of Hash Functions. When rebuilding, DHash heavily relies on the choice of the hash function for the new hash table to resolve hash collisions. Although DHash allows programmers to choose arbitrary hash functions, in practice, we suggest that they first utilize some widely-used non-cryptographic hash functions (e.g., Jenkins' hash function [31]) that are fast and have been empirically proved to be "approximately" universal by changing their hash seeds [32].

If the target buckets are still unacceptably long after rebuilding, programmers are suggested to adopt a *universal hashing* approach [5]. In universal hashing, the system first creates a set of hash functions that are carefully designed, and programmers randomly select one hash function from the set for each hash table. Universal hashing guarantees that for the same group of nodes, two randomly-selected hash functions (one for the old hash table and another for the new one) are extremely unlikely to both cause bad hashing performance [5].

Overall, the proposed strategy for selecting new hash functions can provide good average-case hashing performance for DHash , even if the existing hash function cannot evenly distribute the incoming data, or if an adversary has compromised the existing hash function. Specifically, for any existing nodes (including those in the target buckets), the expected length of the list in the new hash table that the node hashes to is at most the load factor $\alpha = n/m$, where n and m are the total number of the existing nodes in DHash and the number of buckets in the new hash table, respectively (Theorem 11.3 of [33]). Moreover, for a rebuilding scenario where the m and q (the number of nodes in the target buckets) are 1,024 and 32, respectively, the probability of at least 4 and 16 out of these q nodes being hashed to the same bucket are $3.34 * 10^{-5}$ and $4.21 * 10^{-37}$, respectively (Theorem 4 in [34]), which shows that by using the universal hashing approach, it is extremely unlikely that the hash collisions in DHash can remain after rebuilding.

Two Implementations of DHash. In this paper, we built and evaluated two versions of DHash: one by using Michael's classic lock-free linked list [14] as the implementation of its buckets, and another by using a concurrent linked list algorithm [35] that provides wait-free lookup operations. To keep our presentation self contained, we take Michael's linked list [14] as an example and discuss how we modify it before using it in DHash in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2022.3151499.

4.2 Data Structures

Algorithm 2 lists the data structures and auxiliary functions of DHash. The main structure, *ht*, is an array of buckets (*bkts []*), with the array length stored in the *nbuckets* field. Each element of *bkts* is fundamentally a pointer to an instance of *list*, an RCU-based lock-free or wait-free set algorithm that provides the API shown in Algorithm 1. The *hash* field is a function pointer to the user-specified hash function. The *ht_new* field remains *NULL* unless a rebuild operation is in progress, in which case it points to the newly-allocated hash table that is going to replace the old one. The global variable *rebuild_cur* points to the node that is currently in the *hazard period* or is equal to *NULL* if there is no such a node. The mutex lock *rebuild_lock* is to serialize attempts to rebuild the hash table. Note that the workload of rebuilding a hash

table can be parallelized by creating a group of rebuilding threads, which will be discussed in Section 4.7.

The helper function *logically_removed(htnp)* checks whether the *LOGIC_RM* bit of the *next* field of the node pointed to by *htnp* has been set. The helper function *ht_alloc ()* creates a new instance of the hash table and allows users to specify its hash function. Note that the hash function, once set, cannot be changed, and is shared among all rebuilding threads and worker threads accessing this hash table instance.

Algorithm 2. Structures and Auxiliary Functions.

```
 9: struct ht {ht *ht_new; long (*hash)(long key); int nbuckets;
    list *bkts[]};
   /* Global variables.                                        */
10: struct node *rebuild_cur;
11: mutex rebuild_lock;
12: #define logically_removed(htnp) (htnp->next &LOGIC_RM)
13: clean_flag(node *htnp, long flag) {
14:     atomic_fetch_and(htnp->next, ~flag);
15: }
16: ht_alloc(ht *htp, int nbuckets, long (*hash)(long key)) {
17:     htp->ht_new := NULL; htp->hash := hash;
18:     htp->nbuckets := nbuckets; htp->bkts := allocate(nbuckets);
19: }
```

4.3 Rebuild Operation

The pseudocode for the rebuild operation is shown in Algorithm 3. Line 22 attempts to acquire the global lock *rebuild_lock*, which serializes concurrent rebuild attempts. Note that the lock *rebuild_lock* is transparent to concurrent regular operations and thus can never block any of them. Once a rebuild operation has the lock, it checks again that the rebuild is still required on line 23. Line 24 allocates a new hash table with the user-specified bucket array size and hash function. Line 25 assigns the new hash table to the *ht_new* field of the old hash table, allowing subsequent operations to access the new hash table. Line 26 performs an RCU synchronization barrier to wait for prior regular operations, which may not be aware of the new hash table, to complete before the rebuild operation continues.

The function *ht_rebuild()* traverses the old hash table and distributes every node to the new hash table (lines 27–44). Before distributing each node, the function *ht_rebuild()* first enter an RCU read-side critical section (line 29) to prevent concurrent delete operations from erroneously reclaiming the node, while *ht_rebuild()* is still accessing it. Then, the global variable *rebuild_cur* points to the node on line 30. The two write barriers on lines 31 and 40 pair with the read barriers in *ht_lookup()* and *ht_delete()*. They together guarantee that *ht_rebuild()*'s updates to *rebuild_cur* and the two hash tables are performed in program order with respect to concurrent regular operations. Note that, for ease of presentation, we omit memory order specifications in the pseudocode. In practice, all accesses to the bucket pointers (e.g., *htbp*), the node pointers (e.g., *htnp*), *ht_new*, and *rebuild_cur* must be made with the specifications of std:: memory_order_acquired or release [36].

Algorithm 3: Rebuild Operation of DHASH.

```
Parameters: nbuckets: Size of the bucket array.
            hash: User-specified hash function.
20: Local variables: struct node *htnp, *htbp, *htbp_new;
    struct ht *htp_new;
21: void ht_rebuild(ht *htp, nbuckets, hash) {
22:   if ( trylock(rebuild_lock) != SUCCESS ) return -EBUSY;
23:   if ( ! rebuild_is_required() ) return -EPERM;
24:   htp_new := ht_alloc(htp, nbuckets, hash);
25:   htp->ht_new := htp_new;
    /* Wait for operations not aware of htp_new .  */
26:   synchronize_rcu();
27:   for each bucket htbp in htp {
28:       for each node htnp in htbp {
29:     rcu_read_lock();
30:     rebuild_cur := htnp;
31:     smp_wmb();
32:     key := htnp->key;
33:     if (list_delete(htbp, key, IN_HAZARD)!= SUCCESS)//Hflag
34:         continue;
35:     clean_flag(htnp, IN_HAZARD)/* unHflag          */;
36:     htbp_new := htp_new->bkts[htp_new->hash(key)];
37:     if list_insert(htbp_new, htnp) != SUCCESS {
38:         call_rcu(htnp, free_node);
39:     }
40:     smp_wmb();
41:     rebuild_cur := NULL;
42:     rcu_read_unlock();
43:   }
44:   }
    /* Wait for operations accessing nodes via
    htp->bks[] .                                          */
45:   synchronize_rcu();
46:   htp_tmp := htp;  htp := htp_new;
   /* Wait for operations accessing old hash table.  */
47:   synchronize_rcu();
48:   unlock(rebuild_lock);
49:   free(htp_tmp);
50:   return SUCCESS;
51: }
```

Line 33 deletes the node from the old hash table. The function *list_delete()* receives a third argument *IN_HAZARD*, indicating that the node with the matching key will be deleted from the old hash table, but its memory will not be reclaimed. If this delete operation fails, which implies that the node has been deleted by other concurrent delete operations since the reference to the node was fetched on line 28, the rebuild operation skips this node (line 34). Otherwise, the node has been physically removed from the old hash table. It is guaranteed that *rebuild_cur* points to this physically-removed node, because once rebuild operations are in progress, no new node (possibly with the identical key) can be inserted into the old hash table (discussed in detail in Section 4.6). Line 35 then prepares the node for reuse by cleaning the *IN_HAZARD* bit and the *ptr* part of its *next* field. Note that if the *LOGIC_RM* bit has been set by another concurrent delete operation, it remains. Since the node can be logically removed by other concurrent delete operations, *clean_flag()* uses an atomic primitive (line 13). Then, line 37 inserts the node into the proper bucket of the new hash table. This insertion generally succeeds. It fails

only if the node referenced by the global pointer *rebuild_cur* is logically deleted by a concurrent delete operation (discussed in Section 4.5), and then a new node with identical key is inserted into the new hash table before line 37 is performed. In this case, line 38 invokes *call_rcu()*, which reclaims the node after currently unfinished operations pointing to this node have been completed. After the node has been inserted into the new hash table, the global pointer *rebuild_cur* is set back to *NULL*.

After distributing the existing nodes in the old hash table, line 45 waits for prior unfinished operations, which may be holding references to the distributed nodes. Line 46 sets the new hash table as the current one, and line 47 waits for prior unfinished operations that may be holding references to the old hash table. Then, line 48 releases the global lock, line 49 frees the old hash table, and finally, line 50 returns success.

For each iteration, *ht_rebuild()* deletes a node from the old hash table and then inserts it into the new hash table, reusing the node's memory. One potential issue with this distributing mechanism is that it may redirect lookup operations traversing the old hash table to the wrong lists. For example, suppose that a lookup operation is traversing a hash bucket in the old hash table and is pointing to the node α. At the same time, the rebuild operation distributes α by inserting it into the new hash table. This can redirect the lookup operation to the bucket in the new hash table, and result in a false negative if the node with the matching key is at the bottom of the linked list in the old hash table. There are two solutions to this problem.

(1) Each node contains an extra integer field called *bkt_id*, which records the *id* of the bucket where the node is stored. Before traversing a list, a lookup operation first reads and remembers the id of the bucket. Each time the lookup operation moves to a new node, it compares the value stored in the *bkt_id* field with the stored bucket id, and starts over if they are not equal. This approach introduces negligible performance overhead because accessing the *bkt_id* field does not involve any atomic instructions. The shortcoming of this approach, however, is that lookup operations may start over when a rebuild operation is distributing nodes. Note that, even though lookup operations may start over, they will not be blocked because when rebuilding, the length of the lists in the old hash table is limited, and therefore the lookup operations can complete in a finite number of steps.

(2) After a node has been deleted from the old hash table, and before this node is inserted into the new hash table, the rebuild operation invokes a *synchronize_rcu()* barrier and waits until all of the prior regular operations complete and leave their RCU read-side critical sections. Even though this approach can slow down the rebuild operation, it saves the *bkt_id* field in each node and prevents lookup operations from starting over. Given that rebuild operations should be relatively infrequent, we believe this approach is a valuable option for use cases where extremely fast lookup operations are required.

Overall, for DHash implementations that need to provide extremely fast lookup operations, the second solution best meets the requirements. Otherwise, the first one is the solution of choice. The two implementations of DHash presented in this paper (discussed in Section 6.1) use the first solution.

4.4 Lookup Operation

The pseudocode for the lookup operation is presented in Algorithm 4. This function first searches for the specified *key* value in the old hash table (line 54). If a node with the matching key can be found, a pointer to the node is returned (line 54). Otherwise, line 55 checks whether a rebuild operation is in progress, and line 55 returns *-ENOENT* if rebuild operations are absent. The two read barriers on lines 56 and 61 pair with the two write barriers in *ht_rebuild()*. Line 58 continues the lookup operation by checking the node pointed to by the global pointer *rebuild_cur*. Recall that *rebuild_cur* always points to the node that is currently in its *hazard period*. If the node pointed to by *rebuild_cur* matches, and if the *LOGIC_RM* bit of the *next* field of this node has not been set, which means that the node has not been logically deleted by concurrent delete operations, line 59 returns a pointer to the node. Otherwise, *ht_lookup()* continues by searching the new hash table and returns a pointer to the node if the function *list_find()* succeeds (line 64).

Algorithm 4: Lookup Operation of DHash.

```
Local variables: struct node *cur, *htbp, *htbp_new; struct ht
                 *htp_new; struct snapshot ss;
52: node *ht_lookup(ht *htp, long key) {
53:   htbp := htp->bkts[htp->hash(key)];
54:   if (list_find(htbp, key, &ss) = SUCCESS) { return ss.cur };
55:   if (htp->ht_new = NULL) { return -ENOENT };
56:   smp_rmb();
57:   cur := rebuild_cur;
58:   if cur and (cur->key = key) and !logically_removed(cur) {
59:     return cur;
60:   }
61:   smp_rmb();
62:   htp_new := htp->ht_new;
63:   htbp_new := htp_new->bkts[htp_new->hash(key)];
64:   if (list_find(htbp_new, key, &ss) = SUCCESS) { return ss.cur };
65:   else return -ENOENT;
66: }
```

Algorithm 4 shows that a lookup operation first searches for the node with the matching key in the old hash table (line 54), then checks the node referenced by the global pointer *rebuild_cur* (line 58), and finally searches in the new hash table (line 64). This manipulation order guarantees that lookup operations can always find the node. That is, the following lemma holds:

Lemma 1. *If DHash contains a node α with the key value of k, the function ht_lookup(k) can find α, no matter if a rebuild operation is in progress.*

Proof. Obviously, if there are no rebuild operations, the node α resides in the only hash table. The function *ht_lookup(*K*)* can find the node in the only hash table (lines 53 - 54).

We then prove that *ht_lookup(*K*)* can find the node when a rebuild operation is in progress. Note that we assume that the set algorithm used as the implementation of hash table buckets is linearizable, which is the case for most existing non-blocking set algorithms. Therefore, in the following proof, the three list operations, *list_find()*, *list_insert()*, and *list_delete()*, can be treated as atomic operations that

take place as "point" events [14], [23] in the execution history of DHASH. The code snippet to distribute a node is shown on lines 30–41 in Algorithm 3. We use $write_{rebuild}(rebuild\_cur, \alpha)$ to denote the event in which the thread running the rebuild operation (henceforth *rebuild thread* for short) writes the address of the node α to the global variable *rebuild_cur* (line 30), and use $delete_{rebuild}(old, \alpha)$ and $insert_{rebuild}(new, \alpha)$ to denote the events in which α is deleted from and inserted into the old and the new hash tables, respectively (lines 33 and 37). Similarly, we use $lookup_{lookup}(old, \alpha)$ and $lookup_{lookup}(new, \alpha)$ to denote the events in which the thread running lookup operations (henceforth *lookup thread* for short) searches for the node α in the old and the new hash tables, respectively (lines 54 and 64). We use $lookup_{lookup}(rebuild\_cur, \alpha)$ to denote the event in which the lookup thread checks the node pointed to by *rebuild_cur* (line 58). In the following proof, since the rebuild thread is the only thread that performs write, delete, and insert operations, and the lookup thread is the only thread that performs lookup operations, we omit the thread symbol without introducing any ambiguity. For brevity, we use the acronym *rbc* to stand for *rebuild_cur*. One event e_1 precedes another event e_2, written $e_1 \prec e_2$, if e_1 occurs at an earlier time.

By inspecting the code of *ht_rebuild()* in Algorithm 3, we get the following event sequence

$$write(rbc, \alpha) \prec delete(old, \alpha) \prec insert(new, \alpha) \prec write(rbc, \perp) \tag{1}$$

By inspecting the code of *ht_lookup()* in Algorithm 4, we get the following event sequence

$$lookup(old, \alpha) \prec lookup(rbc, \alpha) \prec lookup(new, \alpha) \tag{2}$$

When the rebuild and the lookup threads are simultaneously accessing the node α, there are three types of interleaving between these two threads:

- $lookup(old, \alpha) \prec delete(old, \alpha)$, which implies that the lookup thread searches for the node α before the rebuild thread starts distributing the node. Thus, α can be found in the old hash table, and the lookup operation can return a pointer to α on line 54.
- $insert(new, \alpha) \prec lookup(new, \alpha)$, which implies that the lookup thread searches for the node α after it has been inserted into the new hash table by the rebuild thread. Thus, α can be found in the new hash table, and the lookup operation can return a pointer to α on line 64.
- $delete(old, \alpha) \prec lookup(old, \alpha) \prec \cdots \prec lookup(new, \alpha) \prec insert(new, \alpha)$, which implies that the lookup thread searches for the node α which is in the *hazard period*. Combined with the event sequences 1 and 2, we get the following event sequence:

  $write(rbc, \alpha) \prec delete(old, \alpha) \prec lookup(old, \alpha) \prec lookup(rbc, \alpha) \prec lookup(new, \alpha) \prec insert(new, \alpha) \prec write(rbc, \perp)$

  It follows that:

  $write(rbc, \alpha) \prec lookup(rbc, \alpha) \prec write(rbc, \perp)$

  Once the global pointer *rbc* is set to point to the node α it remains. Hence the lookup thread can find α via *rbc* and can return a pointer to it on line 59.

Overall, if there is a node with the matching key in DHASH, it is guaranteed that the function *ht_lookup(k)* can find the node and return a pointer to it, no matter if a rebuild operation is in progress. □

4.5 Delete Operation

Recall that when DHASH is rebuilding, a node can (1) resides in either the old or the new hash table, or (2) is in the *hazard period* and is referenced by the global pointer *rebuild_cur*. Therefore, the challenges that the delete operation of DHASH must address are (1) how to find the node, and then (2) how to delete this node which is concurrently being distributed from the old hash table to the new one by the rebuild operation.

To address the first challenge, the delete operation uses the same manipulation order as *ht_lookup()* (shown in Algorithm 4). That is, the delete operation first searches for the node with the matching key in the old hash table, then checks the node referenced by the global pointer *rebuild_cur*, and finally searches in the new hash table. Lemma 1 guarantees that if a node with the matching key is in DHASH, the function *ht_delete()* can always find this node.

To address the second challenge, DHASH adopts a classic, lightweight mechanism, by separating the deletion of a node into two stages: *logical* and *physical* deletions [14], [37]. The first stage is to mark a node to prevent subsequent lookup operations from returning this node, and to prevent subsequent insert/delete operations from inserting/deleting nodes after this node. Marking a node is notably by using a *cas* operation that sets the least significant bit (i.e., the *LOGIC_RM* in this paper) of the *next* field of the node. The second stage, which is typically performed by subsequent lookup operations, is to physically remove the node from the list by swinging the next pointer of the previous node to the next node in the list and then reclaiming the node memory.

The pseudocode for the delete operation is shown in Algorithm 5. The function *ht_delete()* first attempts to delete the node from the old hash table on line 70. Then, it continues by checking if a rebuild operation is in progress on line 73. The two read barriers on lines 74 and 83 pair with the two write barriers in *ht_rebuild()*. If a rebuild operation is in progress, *ht_delete()* checks the node referenced by *rebuild_cur*, and attempts to delete the node by setting the *LOGIC_RM* bit if the node has the expected key value (line 79). Since the rebuild operation can simultaneously clean the *IN_HAZARD* bit (line 35) and/or update the *ptr* field (line 37) of this node, *ht_delete()* uses a loop that repeatedly sets the *LOGIC_RM* bit by using a *cas* instruction, which can fail at most two times. If other concurrent delete operations have successfully deleted this node, *ht_delete()* skips this node (line 78). The delete operation continues by attempting to delete the node with the matching key from the new hash table (line 85). If the delete operation fails, line 87 returns *-ENOENT*, indicating that no node with the matching key can be found in DHASH.

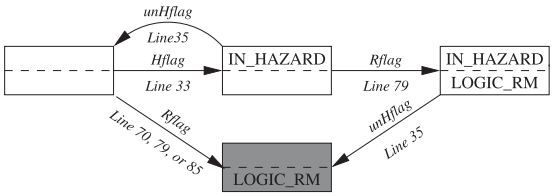


Fig. 2. Sequence of changes a node's *flag* field can go through. The labels in the boxes show the values of the *flag* field of the node. Each of the transitions corresponds to a successful *cas* instruction, and the label shows the specific *cas* instruction along with its line number in the pseudocode.

Algorithm 5: Delete Operation of DHash.

```
67: Local variables: struct node *cur, *htbp, *htbp_new;
    struct ht *htp_new;
68: int ht_delete(ht *htp, long key) {
69:     htbp := htp->bkts[htp->hash(key)];
70:     if (list_delete(htbp, key, LOGIC_RM) = SUCCESS)/*Rflag*/
71:         return SUCCESS;
72:     htp_new := htp->ht_new;
73:     if (htp_new = NULL) return -ENOENT;
74:     smp_rmb();
75:     cur := rebuild_cur;
76:     while cur and (cur->key = key) {
77:         cur_next := cur->next;
78:         if (cur_next &LOGIC_RM) break;
79:         if (cas(cur->next, cur_next, cur_next | LOGIC_RM))
            /*Rflag*/
80:             return SUCCESS;
81:         cur := rebuild_cur;
82:     }
83:     smp_rmb();
84:     htbp_new := htp_new->bkts[htp_new->hash(key)];
85:     if (list_delete(htbp_new, key, LOGIC_RM) =
        SUCCESS)/*Rflag*/
86:         return SUCCESS;
87:     return -ENOENT;
88: }
```

To prove that the delete operation of DHash can successfully delete the expected node, while a rebuild operation is distributing this node, we illustrate the possible values of the two least significant bits of the *next* field (the *flag* field henceforth) of a node and the sequence of changes the node's *flag* field can go through in Fig. 2. The labels in the boxes show the values of the *flag* field, and the labels of the transitions show the corresponding *cas* instructions and line numbers in the pseudocode. For example, the right-most box shows that both the *IN_HAZARD* and *LOGIC_RM* bits have been set, indicating that while the node is being distributed by the rebuild operation (between lines 33 and 35), a delete operation finds it via the global pointer *rebuild_cur* and successfully deletes it on line 79. The *flag* field of a node is initially clean (the left-most box) and can only be changed by the following *cas* operations. (1) To distribute the node, the rebuild operation sets the *IN_HAZARD* bit of the node on line 33, and after physically removing this node out of the old hash table, cleans the *IN_HAZARD* bit on line 35. We refer to these two *cas* steps as *Hflag* and *unHflag*, respectively. (2) To delete the node, a delete operation sets the *LOGIC_RM* bit of the node on either line 70, 79, or 85. We refer to these *cas* steps as *Rflag*.

By examining the pseudocode in Algorithms 3 and 5, we can see that these *cas* operations proceed in the orderly way shown in Fig. 2, and that the sequence of changes shown in Fig. 2 reflects all possible changes to the *flag* field of the node. For example, each time a node is distributed from the old hash table to the new one, its *flag* field walks through the top left circuit (transitions labeled *Hflag* and *unHflag*). Fig. 2 illustrates that the following statements are true for each node.

- No matter if the *IN_HAZARD* bit has been set, exactly one delete operation can succeed, by setting the *LOGIC_RM* bit of the node (via one of the two *Rflag* CASes).
- If the *IN_HAZARD* bit is set, it is eventually cleaned via one of the two *unHflag* CASes, and the *unHflag* CASes leave the *LOGIC_RM* bit untouched.
- Once the *LOGIC_RM* bit is set, it remains; a node that has been deleted eventually reached the state represented by the shaded box in Fig. 2.

Moreover, by examining the code, we can see that once a node's *LOGIC_RM* bit is set, no concurrent lookup operations can return this node. Instead, they help physically remove this node. With all these facts, we get the following lemma:

Lemma 2. *No matter where the node is (either in one of the hash tables, or is referenced by* rebuild_cur*), a delete operation can successfully delete it by setting its* LOGIC_RM *bit.*

Overall, we can prove that the following lemma holds:

Lemma 3. *If DHash contains a node α with the key* k*, the function ht_delete(k) can delete the node α by successfully setting its* LOGIC_RM *bit, no matter if a rebuild operation is in progress.*

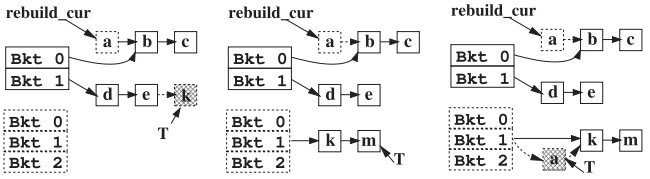
Proof. The function *ht_delete(k)* fundamentally performs a lookup operation along with a *logical deletion* if the node can be found. Lemma 1 guarantees that if a node with the required key is in DHash, the function *ht_delete()* can always find this node. Moreover, Lemma 2 guarantees that once the node can be found, the delete operation can successfully delete it. □

Note that Fig. 2 also allows us to naturally choose the linearization points for delete operations: each successful delete operation is linearized at the successful *Rflag* CAS.

4.6 Insert Operation

The basic idea of DHash's insert operation is as follows. When there are no rebuild operations, which is the common case, DHash inserts new nodes into the only (i.e., old) hash table. Otherwise, new nodes are inserted into the new hash table. (For ease of presentation, we refer to insert operations that add a node with key k into the old and the new hash table as *IO(k)* and *IN(k)*, respectively.)

However, simply applying this strategy in practice might lead to concurrency issues because when rebuild operations are in progress, *IO(k)* and *IN(k)* may exist simultaneously, such that some synchronization mechanisms are required to



(a) Thread T performs IO(k) but fails, because a rebuild operation is in progress.

(b) Thread T then performs IN(k) and succeeds. Subsequent insertions go to the new table.

(c) Thread T attempts to perform IN(a) but fails, because key a has been in DHASH.

Fig. 3. The interaction between hash table insert and rebuild operations. Thread T attempts to insert k, m, and then a, while a rebuild operation is in progress. Shaded boxes indicate that the corresponding insertion attempts failed.

prevent *IO(k)* and *IN(k)* from inserting duplicate nodes in DHASH.

To that end, we adopt a lightweight and conceptually straightforward synchronization strategy that consists of two aspects. (1) Once a rebuild operation starts, *IO(k)* stops inserting nodes into the old hash table. Instead, *IO(k)* starts over and then attempts to insert the node into the new hash table (i.e., it becomes an *IN(k)*). (2) An *IN(k)* first checks if any node with key *k* already exists in the old hash table or is currently referenced by *rebuild_cur*, before attempting to insert its own node into the new hash table. Fig. 3 illustrates how DHASH's insertion works. The initial state of the hash table is the same as that in Fig. 1b, and thread *T* attempts to insert k, m, and then a into DHASH. We assume that the rebuild operation starts while thread *T* is performing IO(k), such that IO(k) will fail and become an IN(k), shown in Fig. 3a. Thread *T* is then aware of the rebuild operation, such that subsequent insertions (e.g., m) directly occur in the new hash table, shown in Fig. 3b. The insertion of node a fails, because before performing IN(a), thread *T* performs a lookup operation which returns an error message indicating that another node with identical key value has been in DHASH, shown in Fig. 3c.

Specifically, we implement the function *list_insert_dcss()*, a variant of the standard insert operation of the underlying buckets discussed in Algorithm 1, that allows us to insert a node into the underlying bucket only if rebuild operations are absent. The function *list_insert_dcss()* returns *-EADDR1* if a rebuild operation started before it linearizes; otherwise, it works as if it were the standard *list_insert()* (i.e., it returns *SUCCESS* if the node is successfully inserted into the list, and *-EEXIST* if such a node already existed in the bucket.)

The implementation of the function *list_insert_dcss()* is similar to that of the standard *list_insert()*, except that the statement where a successful *list_insert()* operation linearizes, which is notably a *cas* or write instruction, is replaced with Harris *et al.*'s lock-free double-compare-and-single-swap (*dcss*) primitive [38]. The *dcss* primitive is implemented from normal *cas* instructions, and takes five arguments: two addresses, two expected values, and one new value, and can atomically (1) read the two memory addresses, (2) check if they contain the expected values, and (3) if so, write the new value into the second address. Therefore, by setting the first memory address and the expected value in this address to *htp->htp_new* and *NULL*, respectively, the function *list_insert_dcss()* can successfully insert the node into the list only if rebuild operations are absent. Once a rebuild operation has started, and the *htp->htp_new* field has been set to point to the newly-allocated hash table on line 25, *list_insert_dcss()* will fail and start over.

The use of the *dcss* primitive is motivated by Arbel-Raviv and Brown's non-blocking range query algorithms [39]. Appendix A, available in the online supplemental material shows the implementation of *list_insert_dcss()* for Michael's classic lock-free linked list algorithm [14].

Algorithm 6: Insert Operation of DHASH.

```
 89: Local variables: node *htnp, *htbp, *htbp_new; ht *htp_new;
 90: int ht_insert(ht *htp, long key) {
 91:     retry:
 92:     htnp := allocate_node(key);
 93:     htp_new := htp->ht_new;
 94:     if htp_new = NULL{
           /* IO(k)                                                  */
 95:       htbp := htp->bkts[htp->hash(key)];
 96:       ret := list_insert_dcss(&htp->htp_new, NULL, htbp, htnp) ;
 97:       if (ret = SUCCESS) return SUCCESS ;
 98:       else if (ret = -EADDR1) {free(htnp); goto retry (line 91);}
 99:     }
100:     else{
           /* IN(k)                                                  */
101:       if (ht_lookup(htp, key) = SUCCESS) {free(htnp); return
          -EEXIST;}
102:       htbp_new := htp_new->bkts[htp_new->hash(key)];
103:       if (list_insert(htbp_new, htnp) = SUCCESS)
104:           return SUCCESS;
105:     }
106:     free(htnp);
107:     return -EEXIST;
108: }
```

Algorithm 6 shows the pseudocode for the insert operation of DHASH. The function first allocates a new node, initializes it (line 92), and then checks if a rebuild operation is in progress (line 94). For an *IO(k)*, the function *list_insert_dcss()* will fail if a rebuild operation starts during the time gap between lines 93 and 96 of this insert operation. In this case, *list_insert_dcss()* returns an error message *-EADDR1*. Upon receiving this message, *IO(k)* starts over (line 98), and in the subsequent insertion attempt, it can see the new hash table (line 93) and will insert the node into it. That is, it becomes *IN(k)*. Note that the RCU mechanism (line 26) prevents another rebuild operation from starting before this *IO(k)* completes, such that an *IO(k)* restarts at most once. That is, the following lemma holds:

Lemma 4. *Once a rebuild operation has started,* IO(k) *will fail, start over, and then attempt to insert the node into the new hash table; that is, it becomes* IN(k).

An *IN(k)* first performs a lookup operation on line 101, and then attempts to insert the node into the new hash table (line 103). By examining the code, we can prove that the following lemma holds:

Lemma 5. *When* IN(k) *attempts to insert a node α with key k into the new hash table on line 103, it is guaranteed that (1) the old hash table does not contain any node with key k, and (2) the node referenced by* rebuild_cur *does not have the key k.*

Proof. When *IN(k)* performs the lookup operation on line 101, which implies that a rebuild operation has started, Lemma 4 guarantees that no new nodes will be inserted into the old hash table. Moreover, if any existing node in DHash has the key k, Lemma 1 guarantees that this lookup operation can find it, and return the error message *-EEXIST*. Since a new node with key k can be inserted into the new hash table during the time gap between lines 101 and 103 of this insert operation, we can only guarantee that the node with key k cannot appear in the old hash table and cannot be referenced by *rebuild_cur* when *IN(k)* performs the line 103. □

With Lemma 4 and 5, now we can prove that the following lemma holds:

Lemma 6. *If DHash does not contain a node with key k, the function* ht_insert(k) *can successfully insert a node α with key k in DHash; otherwise, the error message* -EEXIST *is returned without changing the hash table, no matter if a rebuild operation is in progress.*

Proof. Obviously, if there are no rebuild operations, the value of *htp->htp_new* is always equal to *NULL*, such that the function *ht_insert(k)* always attempts to insert the node α into the only hash table (lines 94 – 98).

When rebuild operations are in progress, *IO(k)* and *IN (k)* can exist simultaneously. However, Lemma 4 guarantees that *IO(k)* will start over and become *IN(k)*. Therefore, the function *ht_insert(k)* always attempts to insert the node α into the new hash table. Moreover, Lemma 5 guarantees that a duplicate node with key k can neither exist in the old hash table nor be referenced by *rebuild_cur*.

Overall, DHash always attempts to insert new nodes into either the old or the new hash table, depending on whether rebuild operation are absent, such that the insert operation of DHash complies with the standard specification for the insert operation of a normal hash table. □

Note that the original implementation of the *dcss* primitive [38] uses a helping mechanism, which can lead to poor performance because each *dcss* operation needs to dynamically allocate (and later, free) an *Info* object that holds the required information for helping. To address this performance issue, we applied the following optimizations in our implementation. (1) Motivated by Arbel-Raviv and Brown's research results [40], the *Info* objects for the *dcss* operation are pre-allocated and reused. Additional fields such as sequence numbers and flags are used to prevent insert operations from erroneously reading improper information from these shared, global objects [40]. (2) The *Info* objects are carefully placed and properly aligned in memory to avoid cache thrashing [41], when multiple insert operations update them simultaneously. (3) Lookup operations that might access these objects utilize hardware prefetch unit as much as possible by prefetching the possible objects before accessing their contents. Experimental results show that our *dcss* implementation introduces negligible overhead to DHash in terms of throughput and operation latency.

4.7 Parallelizing Rebuild Operation

Even though the rebuild operation of DHash is lightweight (demonstrated in Section 6.3), it is expected to be completed as soon as possible. To that end, we parallelize the rebuild operation by leveraging the *thread-level parallelism* [41].

The basic idea behind our parallelizing strategy is that in distributing the nodes of the old hash table, different buckets can be processed in parallel, since they are mainly independent of one another. Specifically, when rebuilding, the rebuild operation of DHash create a group of n threads (referred to as *rebuilding threads*), each of which is bind to a dedicated CPU core. We then divide the bucket range of the old hash table to n interleaved regions, and then assign different regions to different rebuilding threads. That is, the bucket b is assigned to the rebuilding thread i if $b \% n = i$. Each rebuilding thread takes the argument of a region of buckets, and distributes the nodes in these buckets to the new hash table by using the program logic shown on lines 27 – 44 in Algorithm 3. The only modification is that *rebuild_cur* becomes a global, per-thread variable; each rebuilding thread has its own *rebuild_cur*, which points to the node that is being distributed by this thread and is in the *hazard period*. Rebuilding threads may simultaneously insert nodes into the same bucket in the new hash table, which is synchronized by the lock-free or wait-free set algorithm used as the implementation of the buckets.

Accordingly, each time a lookup, insert, or delete operation wants to check if a node with the matching key is in the *hazard period*, it checks the *rebuild_cur* pointers of all rebuilding threads. For example, for the function *ht_lookup()* (Algorithm 4), the only modification is that the code snippet on lines 57 – 60 will be included in a for-loop statement, and that each iteration checks the node that is being pointed to by one rebuilding thread's *rebuild_cur* pointer. One benefit of our parallelizing strategy is that except for letting the lookup, insert, and delete operations to check a group of *rebuild_cur*, the control flow of these operations remains, significantly simplifying the proof of the correctness of DHash when a parallelized rebuild operation is running.

5 Correctness

In this section, we prove that DHash is linearizable and implements an node set object in a lock-free or wait-free manner, depending on the implementation of the hash table buckets.

Set Semantics. We first define the set H of nodes of DHash in any given state as follows: if rebuild operations are absent, H is composed of the non-logically-removed nodes in the only (i.e., old) hash table; otherwise, H is the union of (1) the non-logically-removed nodes in both the old and the new hash tables, and (2) the non-logically-removed nodes that are referenced by the *rebuild_cur* pointers.

We then claim that DHash complies with the abstract set semantics, and we use the standard sequential specification defined in [33] that includes the following three functions:

- The *ht_lookup(k)* operation returns a pointer to the node with key k in H, *-ENOENT* otherwise.
- The *ht_insert(k)* operation returns *SUCCESS* if a node with key k is successfully inserted into H, and *-EEXIST* if such a node already existed in H.
- The *ht_delete(k)* operation returns *SUCCESS* if the node with key k was successfully deleted from H, and *-ENOENT* if such a node cannot be found in H.

We first prove that DHASH is linearizable to this sequential specification. That is, each of DHASH's regular operations has specific linearization points, where the operation takes effect and maps the operation in our concurrent implementation to sequential operations so that the histories meet the specification.

Recall that DHASH is modular, and that any linearizable lock-free or wait-free set algorithm (e.g., linked lists) that implements the API listed in Algorithm 1 can be used as the implementation of DHASH's bucket. The execution histories of DHASH include sequences of list operations (*list_find*, *list_insert*, and *list_delete*) which can be treated as atomic operations because they are linearizable objects [14], [23]. Moreover, we prove in Appendix A, available in the online supplemental material that the function *list_insert_dcss()* is linearizable.

Linearization Points. For each of DHASH's regular operations, linearization points exist within its execution interval, so that at this point, it atomically reads or changes the node set H.

(1) Every lookup operation that finds the node with the expected key via *rebuild_cur*, or in the old or the new hash table is linearized on line 57, 54, or 64, respectively. When rebuild operations are absent, every lookup operation that returns *-ENOENT* is linearized on the first invocation of *list_find()* (line 54). In contrast, when rebuild operations are in progress, defining the linearization point for a lookup operation that returns *-ENOENT* is a bit intricate, because there is a time gap between *ht_lookup()* searches the old hash table (on line 54) and checks the existence of rebuild operations (on line 55), such that a new node with the desired key might be inserted into the old hash table after it has been searched. Therefore, in this case, we linearize the unsuccessful lookup operation within its execution interval at the earlier one of the following points: (1) the point immediately before a new node with the expected key is inserted into the old hash table, and (2) the point where the search on the new hash table returns *-ENOENT*. This linearization point definition is motivated by Heller *et al.*'s classic concurrent set implementation [35].

(2) Similar to the lookup operation, every successful delete operation that finds the node with the required key via *rebuild_cur* takes effect on line 79. For other successful cases, the delete operation linearizes in either of the two invocations of *list_delete()* (line 70 or 85). When rebuild operations are absent, every delete operation that returns *-ENOENT* linearizes on the first invocation of *list_delete()* (line 70). When rebuild operations are in progress, we linearize the unsuccessful delete operation within its execution interval at the earlier one of the following points: (1) the point immediately before a node with the expected key is inserted into the old hash table, and (2) the point where the invocations of *list_delete()* on line 85 returns *-ENOENT*.

(3) Every successful *IN(k)* linearizes in the invocation of *list_insert()* on line 103. Unsuccessful *IN(k)* linearizes on line 101 or 103, depending on where the existing node with the required key is found. *IO(k)* that fails because of the existence of concurrent rebuild operations (i.e., the return value of *list_insert_dcss()* is equal to *-EADDR1*) will attempt to insert the node into the new hash table and linearizes as an *IN(k)* does. Otherwise, *IO(k)* linearizes in the invocation of *list_insert_dcss()* (line 96).

Given the above definition, it is straightforward to deduce from Lemmas 1, 3, and 6 that for each of DHASH's regular operations, within its execution interval, there is a linearization point at which the operation reads or modifies the abstract state H, according to the specified set semantics. That is, the following lemma hold:

Theorem 1. *DHASH is a linearizable implementation of a sequential set object.*

Progress Guarantee. The lookup, insert, and delete operations of DHASH are non-blocking and can provide lock-free or wait-free progress guarantee, depending on the underlying set algorithm used.

(1) A lookup operation *ht_lookup()* invokes the list operation *list_find()* twice. Other statements are regular instructions, which can complete in a finite number of CPU cycles. One corner case is that for some list implementations, the function *list_find()* may need to help other concurrent insert operations, which can be continuously inserting new nodes into the same linked list via the *dcss* primitive (discussed in Appendix A, available in the online supplemental material). However, the progress guarantee of *list_find()* does not degenerate, despite this helping workload, because each time it helps a pending insert operation, it moves forward one node, and this can happen only a limited number of times. Otherwise, a rebuild operation is incurred, preventing subsequent insert operations from inserting nodes into this list in the old hash table.

(2) A delete operation invokes the list operation *list_delete()* twice, and utilizes *call_rcu()* that is non-blocking. As discussed in Section 4.5, the *cas* operation on line 79 fails at most twice. Other statements are regular instructions.

(3) The function *list_insert_dcss()* is a variation of the function *list_insert()*, such that they have the same progress guarantee. An *IO(k)* jumps to the label *retry* and starts over at most once. Therefore, an insert operation invokes either the function *list_insert_dcss()* or *list_insert()* once for most cases, and when there are concurrent rebuild operations, it may invoke both of them once. Other statements are regular instructions.

In summary, DHASH provides a wait-free framework for regular operations. For the implementation of DHASH utilizing Michael's lock-free linked list, the lookup, insert, and delete operations are lock-free. However, since DHASH is modular, programmers can instead choose a wait-free linked list and build their own DHASH that provides a stronger progress guarantee.

Note that the rebuild operation of DHASH could block. Specifically, *ht_rebuild()* serializes concurrent rebuild attempts by using a mutex lock and waits for prior hash table regular operations by using the *synchronize_rcu* barriers. However, this mutex lock is only used by the rebuild operations to serialize concurrent rebuild attempts, and hence can never block concurrent regular operations. Similarly, the RCU barriers in the rebuild operation are used to force a build operation to wait for the existing regular operations. However, they are transparent to regular operations, and cannot block any of them. Moreover, rebuild operations are infrequent, and their speed is not the major concern if they do not noticeably degrade the performance of concurrent regular operations. We thus leave making the helper function *ht_rebuild* non-blocking as our future work.

6 Evaluations

DHash is the first dynamic hash table providing lock-free and wait-free regular operations. Therefore, we need to prove that it is robust, efficient, and scalable. In this section, we demonstrate that (1) DHash's rebuild operation is effective, (2) DHash's rebuild operation is lightweight and does not noticeably degrade system performance in terms of throughput and response time of concurrent regular operations, and (3) DHash's rebuild operation is fast and scalable; that is, users can obtain an approximately linear speedup by creating more rebuilding threads.

6.1 Evaluation Methodology

Hash Table Implementations. We chose Xu's algorithm [18] as representative of dynamic hash tables that maintain two sets of list pointers in each node, and refer to this algorithm as `HT-Dyn-Classic`. Another dynamic hash table widely used is the *rhashtable* algorithm in the Linux kernel [19] (henceforth referred to as `HT-Dyn-Linux`). We chose *HT-Dyn-Linux* as representative of dynamic hash tables that maintain one unordered linked list for each hash bucket. Both *HT-Dyn-Classic* and *HT-Dyn-Linux* use locks to synchronize concurrent insert and delete operations on the same bucket. We chose Shalev and Shavit's algorithm (henceforth referred to as `HT-Resizable`) as representative of resizable hash tables.

To demonstrate the benefits of DHash's modularity, we implemented and evaluated two versions of DHash. *HT-DHash-lf* utilizes Michael's classic lock-free linked list [14] as the implementation of its buckets. The lookup operation of *HT-DHash-lf* needs to start over from the beginning of the list once it encounters *logically* removed nodes. Moreover, we implemented *HT-DHash-wf*, another version of DHash that provides wait-free lookup operations, by utilizing Heller *et al.*'s concurrent linked list algorithm [35]. Note that Heller *et al.*'s linked list provides wait-free lookup operations, but its insert and delete operations are serialized by an optimistic, fine-grained locking mechanism.

In the experiments, we chose Jenkins' hash function [31]. Jenkins' hash function takes a *hash_seed* parameter, and for a benchmarking framework, the hash function with different *seed* values can be regarded as different hash functions. For each regular operation, Jenkins' hash function is invoked to generate a 32-bit hash result for the key, and the hash result modulo the size of the bucket array is used as the index to the bucket array. When rebuilding, we change the *seed* value.

We implemented DHash and the above-mentioned alternatives as user-space programs by using C. For all of the hash tables, optimizations such as cache-line padding are applied. Our benchmarking framework utilizes Desnoyers' user-space implementation of RCU [29]. Specifically, the *QSBR flavor* [29] model is used, such that the lookup operations can perform as fast as single-threaded programs. We compile the code with GCC 7.4.0 on all platforms where Ubuntu 18.04.4 is installed. We use -O3 as our optimization level without any special optimization flags.

Benchmarking Framework. To compare DHash with the alternatives, we built a benchmarking framework for evaluating dynamic hash tables. We give a brief overview of the framework in this section, and refer interested readers to the full version of this paper [42].

The framework consists of a specified number of *worker threads*, each of which performs the workload with the specified distribution of *lookup*, *insert*, and *delete* operations, denoted as `m`. The range of keys `U` is set to ten million to prevent the CPU caches from buffering the whole test set. When a test starts, each worker thread performs an infinite loop. In each iteration, the worker thread randomly selects an operation type according to the specified distribution `m`, randomly chooses a key ranging from 0 to the specified upper bound `U`, and then performs the specified operation. We set the initial size of the bucket array β to 1K. We controlled the average load factor α indirectly by inserting $\alpha * \beta$ nodes in a hash table before starting a test, and by selecting the ratio of insert operations to be equal to that of delete operations.

An *attacker thread* continuously inserts nodes with the keys mapped to a few target buckets of the evaluated hash table, simulating uneven distribution. The attacker thread inserts these nodes at the rate of 300 nodes per second, simulating malicious traffic that uses low bandwidth and is hard to be detected. When a test starts, a *rebuild daemon* is created. Each bucket maintains a per-bucket atomic counter that records the number of nodes in this bucket. A successful insert operation increments this counter, and if it becomes larger than a pre-defined threshold, a signal is sent to the rebuild daemon that in turn, rebuilds the hash table.

We evaluate the hash tables on a Dell PowerEdge server, with 64 GB memory and two Intel Ivy Bridge CPUs, each having 12 cores running at 2.6 GHz. Each CPU has 15 MB shared L3 cache. Each plotted data point, unless specified otherwise, represents the mean value of 20 trials. For performance tests, the standard deviation is denoted by vertical bars, which, however, may be too small to be visible in some figures.

6.2 Effects of Rebuilding

We first evaluate the effects of the rebuild operations. To that end, in this experiment, our benchmarking framework only measures the response time of lookup operations on the target buckets. The statistics of insert and delete operations and the statistics of lookup operations on the non-target buckets will be discussed in Section 6.3. The response time is measured by the worker threads as the time between an operation is invoked and that the hash table returns a result.

In this experiment, we evaluate the hash tables under a *typical hash-table usage pattern* that consists of eight worker threads, with the average load factor α being 16, and with the operation distribution `m` being 5, 5, and 90 for insert, delete, and lookup operations, respectively. The initial and the maximum size of the bucket array are set to 1 K and 64 K, respectively. Once the length of any bucket becomes larger than 32, two times of α, a rebuild request is sent to the rebuild daemon, which rebuilds the hash table by creating a single rebuilding thread.

To better evaluate the effectiveness of the rebuild operations, we further measure `HT-DHash-lf` under the same usage pattern but do not run the attacker thread; that is, we measure how concurrent operations of dynamic hash tables

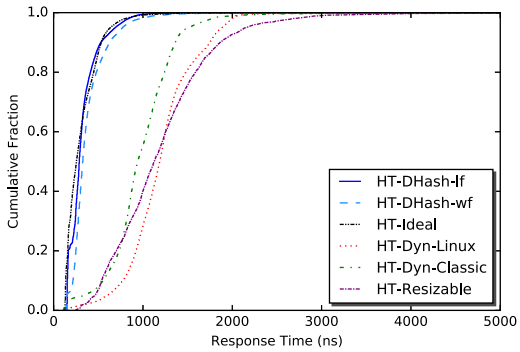


Fig. 4. Response time CDF of lookup operations when rebuilding.

TABLE 1
Response Time Statistics of Lookup Operations When Rebuilding (Nanoseconds)

Hash Table	Median	95%- percentile	99%- percentile	99.9%- percentile
HT-Dyn-Linux	1,211	1,888	2,060	5,492
HT-Dyn-Classic	1,076	1,512	2,038	5,393
HT-Resizable	1,196	2,152	3,003	7,665
HT-Ideal	256	636	944	1,571
HT-DHash-lf	288	692	920	1,546
HT-DHash-wf	328	813	1,361	2,012

behave, when the length of the list of each bucket is roughly equal to the average load factor. This evaluation is the ideal case for a hash table, which we refer to as `HT-Ideal`.

Fig. 4 shows the Cumulative Distribution Function (CDF) of response time, and Table 1 lists the corresponding statistics. We make the following observations. First, once the pre-defined hash function cannot evenly distribute incoming data, DHASH is much more effective in cutting the length of the lists of the target buckets, such that concurrent lookup operations that need to traverse these lists do not experience increased delays. Fig. 4 shows that both versions of DHASH outperform the alternatives in lookup operations' response time. In particular, Table 1 shows that the median response time of `HT-DHash-lf` and `HT-DHash-wf` is 288 and 328 nanoseconds, respectively. In contrast, the alternatives are more than one thousand nanoseconds, about 3 times slower than DHASH. Moreover, the 99.9%-percentile response time of DHASH is under 2 microseconds, but the alternatives reach 5 microseconds, introducing noticeable delays to legitimate users. The reasons for this are as follows. (1) Our distributing mechanism allows DHASH to use lock-free and wait-free lists as the implementation of the buckets. Therefore, the rebuild operation can run in parallel with concurrent regular operations. In contrast, `HT-Dyn-Classic` and `HT-Dyn-Linux` employ a locking mechanism, such that the locks are contended, and the rebuild operation could be suspended, leading to a much longer time period during which concurrent lookup operations must traverse the lists of the target buckets. (2) DHASH is modular, and both `HT-DHash-lf` and `HT-DHash-wf` employ ordered linked lists. In contrast, `HT-Dyn-Linux` can only use unordered linked lists, which causes concurrent lookup operations to traverse many more nodes to get the results.

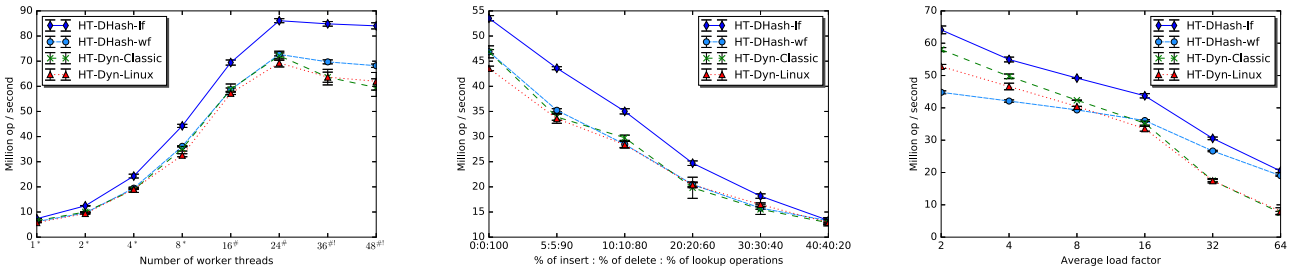
The second observation is that the performance of `HT-Resizable` deteriorates sharply when its pre-defined hash function cannot evenly distribute incoming data. Fig. 4 shows that `HT-Resizable` has the worst tail latencies compared with other hash tables. Moreover, it is interesting that `HT-DHash-lf` even outperforms `HT-Ideal` in some cases. Table 1 shows that the 99.9%-percentile response time of `HT-DHash-lf` is 1,546 nanoseconds, smaller than `HT-Ideal`'s 1,571 nanoseconds.

6.3 DHASH Performance

After evaluating the effectiveness of DHASH's rebuild operation under a typical hash-table usage pattern, we now evaluate the performance of DHASH in a broad range of use cases.

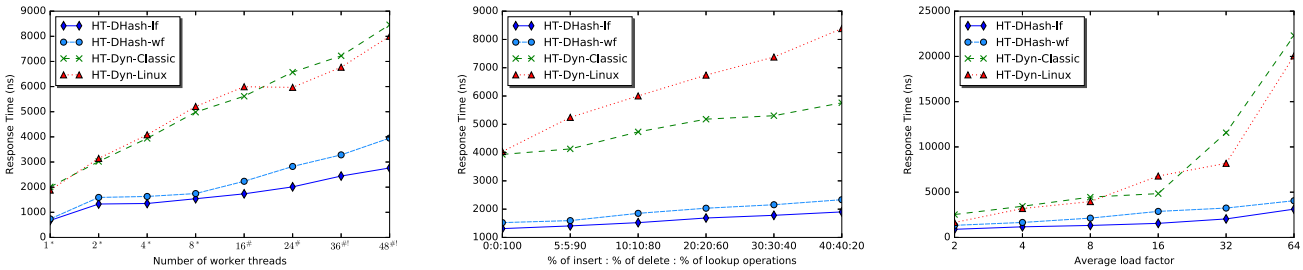
In the experiments, `HT-Resizable` quickly exhausts its bucket array (64 K in maximum) and requests more, and hence the benchmarking framework kills it in a few seconds. We thus omit `HT-Resizable` in the following experiments. The hash table configuration is the same as in Section 6.2, but we vary (1) the number of concurrent worker threads, (2) operation mixes, or (3) average load factors. We make the following observations.

Throughput. First, DHASH outperforms the alternatives at different concurrency levels. Fig. 5a shows that all hash tables have similar throughput results when running with a single worker thread. As the number of worker threads increases, `HT-DHash-lf` outperforms the alternatives. The main reason for this is that to synchronize the rebuild operation and concurrent insert and delete operations, `HT-Dyn-Linux` and `HT-Dyn-Classic` employ per-bucket locks. Even though `HT-DHash-wf` uses an optimistic, fine-grained locking mechanism, it still suffers from the contention on the target buckets. When the server becomes overloaded (the number of worker threads exceeds that of the CPU cores), the throughput of `HT-Dyn-Linux` and `HT-Dyn-Classic` decreases sharply due to the increased contention



(a) At different concurrency levels (b) With different operation mixes (c) With different average load factors

Fig. 5. Throughput (lookup, insert, and delete operations).



(a) At different concurrency levels (b) With different operation mixes (c) With different average load factors

Fig. 6. 99.9%-percentile response time (lookup operations).

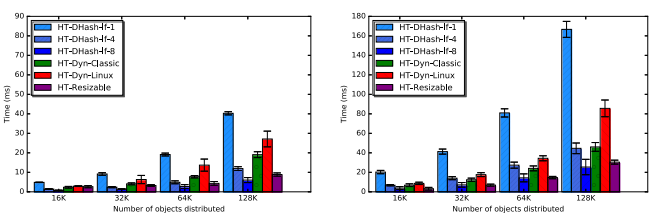
on locks. HT-DHash-lf, in contrast, is much more robust, and there is no noticeable decrease in its throughput.

Second, Fig. 5b shows that as the percentages of insert and delete operations increase, the throughput of all hash tables decreases. However, DHash still outperforms the alternatives, except with the extreme operation mix consisting of 40% of insert, 40% of delete, and 20% of lookup operations. The reason for this is that even for DHash, insert and delete operations are much more costly compared with lookup operations because they must update shared variables and issue memory fence instructions.

Third, DHash outperforms the alternatives with different average load factors. The main reason for this is that HT-DHash-lf employs a lock-free, ordered linked list.

Response Time. When evaluating the throughput, we also measured the corresponding 99.9%-percentile response time of regular operations. We present the results of lookup operations in Fig. 6. Due to lack of space, we omit the results of update operations, which are similar to that of lookup operations, and refer interested readers to the full version of this paper [42].

The response time of DHash's lookup operations increases mainly when the server is overloaded. Fig. 6a shows that the worst 99.9%-percentile response time of HT-DHash-lf is around 3 microseconds, when there are 48 concurrent worker threads. In contrast, HT-Dyn-Linux and HT-Dyn-Classic reach as high as 8 microseconds. Second, the response time of HT-DHash-lf's lookup operations is irrelevant to the operation mixes. Fig. 6b shows that 99.9% of HT-DHash-lf's lookup operations can complete in 2 microseconds, but HT-Dyn-Linux takes more than 8.3 microseconds, when the percentage of lookup operations is reduced to 20%. Third, Fig. 6c shows that as the average load factor increases, the 99.9%-percentile response time of DHash's lookup operations increases slowly. In contrast, the lookup operations of both HT-Dyn-Linux and HT-Dyn-Classic suffer unexpected delays, mainly because of the unordered linked lists used.



(a) 1 worker thread (b) 8 worker threads

Fig. 7. Time required to rebuild hash tables.

6.4 Rebuilding Efficiency

We next measure the efficiency of the rebuild operations; that is, how fast can DHash and the alternatives change their hash functions and rebuild the hash tables. We use the same hash table usage pattern as in Section 6.2, but we vary (1) the total number of nodes in the hash tables, which relates to the workload of rebuild operations, and (2) the number of concurrent worker threads, which causes the rebuild operations to run at different concurrency levels. Since HT-DHash-lf and HT-DHash-wf show the same general trends, for clarity, we only present the results of HT-DHash-lf in this paper. To evaluate our parallelizing strategy, we measure HT-DHash-lf with different numbers of rebuilding threads, and the experimental results are marked with different suffixes. Fig. 7 shows the results. We make the following observations.

First, our parallelizing strategy can achieve an almost linear speedup as a function of the number of rebuilding threads. For example, Fig. 7a shows that it takes the sequential version (HT-DHash-lf-1) 40.3 ms to rebuild a hash table with 64K nodes, and that the time the 4-rebuilding-threads and the 8-rebuilding-threads versions need are 11.9 and 6.0 ms, achieving 3.4x and 6.7x performance speedups, respectively.

Second, for dynamic hash tables, which need to distribute nodes to the new hash tables, the time required to rebuild is almost linear to the number of nodes in the old hash tables. For example, Fig. 7a shows that it takes HT-Dyn-Linux 13.7 and 27.1 ms to respectively rebuild hash tables containing 32K and 64K nodes. This implies that parallelizing rebuild operations is a must have for large dynamic hash tables.

7 Conclusion

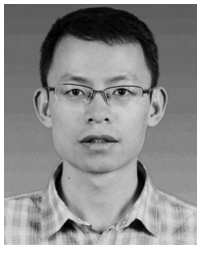
To recover from the emerging algorithmic complexity attacks, this paper presents DHash, one type of hash table that can dynamically change its hash function and rebuild the hash table on the fly. The core of DHash is a novel distributing mechanism that is atomic, non-blocking, and efficient in distributing every node from the old hash table to the new one. DHash is modular and allows programmers to select from existing lock-free and wait-free set algorithms to create their own hash tables. We present the core techniques and demonstrate that DHash is efficient and scalable. Experimental results indicate that DHash is the algorithm of choice for operating systems and applications under service-level agreement (SLA) contracts.

ACKNOWLEDGMENTS

We thank Pedro Ramalhete, Paul E. McKenney, and anonymous reviewers, whose detailed suggestions greatly improved this paper.

REFERENCES

[1] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *Proc. 12th Conf. USENIX Secur. Symp.*, 2003, pp. 29–44.
[2] J. Edge, Denial of service via hash collisions, 2012, Accessed: Feb. 09, 2020. [Online]. Available: https://lwn.net/Articles/474912/
[3] P. Joye, [PHP-DEV] 5.3.9 Release, Hash DoS, 2012, Accessed: Feb. 09, 2020. [Online]. Available: https://lwn.net/Articles/474971/
[4] Microsoft, Vulnerability in ASP.NET Could Allow Denial of Service, 2011, Accessed: Sep. 09, 2021. [Online]. Available: https://docs.microsoft.com/en-us/security-updates/SecurityAdvisories/2011/2659883
[5] L. Carter and M. Wegman, "Universal classes of hash functions," *J. Comput. Syst. Sci.*, vol. 18, pp. 143–154, 1979.
[6] Y. Orton, Hardening Perl's Hash Function, 2013, Accessed: Sep. 09, 2021. [Online]. Available: http://blog.booking.com/hardening-perls-hash-function.html
[7] R. J. Tobin and D. Malone, "Hash pile ups: Using collisions to identify unknown hash functions," in *Proc. Int. Conf. Risks Secur. Internet Syst.*, 2012, pp. 1–6.
[8] N. Bar-Yosef and A. Wool, "Remote algorithmic complexity attacks against randomized hash tables," in *Proc. Int. Conf. Secur. Cryptogr.*, 2007, pp. 117–124.
[9] C. Delimitrou and C. E. Kozyrakis, "Bolt: I know what you did last summer... in the cloud," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2017, pp. 599–613.
[10] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *J. ACM*, vol. 53, pp. 379–405, 2006.
[11] J. Triplett, P. E. McKenney, and J. Walpole, "Resizable, scalable, concurrent hash tables via relativistic programming," in *Proc. USENIX Annu. Tech. Conf.*, 2011, pp. 1–14.
[12] Y. Liu, K. Zhang, and M. Spear, "Dynamic-sized nonblocking hash tables," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2014, pp. 242–251.
[13] P. Fatourou, N. D. Kallimanis, and T. Ropars, "An efficient wait-free resizable hash table," in *Proc. 30th Symp. Parallelism Algorithms Archit.*, 2018, pp. 111–120.
[14] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proc. 14th Annu. ACM Symp. Parallel Algorithms Archit.*, 2002, pp. 73–82.
[15] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *Proc. 14th Annu. ACM Symp. Princ. Distrib. Comput.*, 1995, pp. 214–222.
[16] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," in *Proc. 23rd Annu. ACM Symp. Princ. Distrib. Comput.*, 2004, pp. 50–59.
[17] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank, "Wait-free linked-lists," in *Proc. 17th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2012, pp. 309–310.
[18] H. Xu, "Bridge: Add core IGMP snooping support," 2010, Accessed: Jan. 09, 2021. [Online]. Available: https://marc.info/?l=linux-netdev&m=126798609513579
[19] T. Graf, "Resizable, scalable, concurrent hash table," 2014, Accessed: Jan. 10, 2021. [Online]. Available: https://mail.openvswitch.org/pipermail/ovs-dev/2014-August/286945.html
[20] H. Gao, J. F. Groote, and W. H. Hesselink, "Lock-free dynamic hash tables with open addressing," *Distrib. Comput.*, vol. 18, pp. 21–42, 2004.
[21] T. Maier, P. Sanders, and R. Dementiev, "Concurrent hash tables: Fast and general?(!)," *ACM Trans. Parallel Comput.*, vol. 5, pp. 1–32, 2019.
[22] D. E. Knuth, *The Art of Computer Programming*. London, U.K.: Pearson Educ., 1997.
[23] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.
[24] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, pp. 89–97, 2010.
[25] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, Jun. 2004.
[26] J. Triplett and P. E. McKenney, "Scalable concurrent hash tables via relativistic programming," *Oper. Syst. Rev.*, vol. 44, pp. 102–109, 2010.
[27] IBM, "K42 Operating System," 2006, Accessed: May 09, 2019. [Online]. Available: https://en.wikipedia.org/wiki/K42
[28] P. McKenney, "Introduction to RCU," 2020, Accessed: May 09, 2019. [Online]. Available: http://www.rdrop.com/ paulmck/RCU/
[29] M. Desnoyers, "Userspace RCU," 2012, Accessed: May 09, 2019. [Online]. Available: https://liburcu.org/
[30] P. E. McKenney, "Is parallel programming hard, and, if so, what can you do about it?," 2020, Accessed: Jan. 10, 2021. [Online]. Available: http://www.rdrop.com/~paulmck
[31] B. Jenkins, "A hash function for hash table lookup," 2013, Accessed: Feb. 09, 2020. [Online]. Available: http://www.burtleburtle.net/bob/hash/doobs.html
[32] N. Hua, E. Norige, S. S. Kumar, and B. Lynch, "Non-crypto hardware hash functions for high performance networking asics," *Proc. ACM/IEEE 7th Symp. Archit. Netw. Commun. Syst.*, 2011, pp. 156–166.
[33] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, "Introduction to algorithms," 3rd ed., Cambridge, MA, USA: MIT Press, 2009.
[34] K. Suzuki, D. Tonien, K. Kurosawa, and K. Toyota, "Birthday paradox for multi-collisions," in *Proc. Int. Conf. Inf. Secur. Cryptol.*, 2006, 29–40.
[35] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit, "A lazy concurrent list-based set algorithm," in *Proc. 9th Int. Conf. Princ. Distrib. Syst.*, 2005, pp. 3–16.
[36] C. Spec., "Memory order specification," 2011, Accessed: Feb. 09, 2020. [Online]. Available: https://en.cppreference.com/w/cpp/atomic/memory_order
[37] T. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proc. 15th Int. Conf. Distrib. Comput.*, 2001, pp. 300–314.
[38] T. Harris, K. Fraser, and I. Pratt, "A practical multi-word compare-and-swap operation," in *Proc. 16th Int. Conf. Distrib. Comput.*, 2002, pp. 265–279.
[39] M. Arbel-Raviv and T. Brown, "Harnessing epoch-based reclamation for efficient range queries," in *Proc. 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2018, pp. 14–27.
[40] M. Arbel-Raviv and T. Brown, "Reuse, don't recycle: Transforming lock-free algorithms that throw away descriptors," 2017, *arXiv:1708.01797*.
[41] J. Hennessy and D. Patterson, "Computer architecture - a quantitative approach," 5th ed., San Mateo, CA, USA: Morgan Kaufmann, 1996.
[42] J. Wang, D. Liu, X. Fu, F. Xiao, and C. Tian, "DHash: Dynamic Hash Tables with Non-blocking Regular Operations," 2021, Accessed: Sep. 09, 2021. [Online]. Available: https://junchangwang.github.io/paper/DHash-full.pdf



Junchang Wang received the PhD degree in computer science from the University of Science and Technology of China in 2014. He is currently an assistant professor with the Nanjing University of Posts and Telecommunications. From 2012 to 2013, he was a visiting student with the Department of Computer Science, Yale University. From 2014 to 2016, he was a research director with Jiangsu Future Networks Innovation Institute, China. From 2019 to 2020, he was a visiting scholar with the Department of Computer Science, Chinese University of Hong Kong. His research interests include computer networks, and parallel and distributed computing.

Dunwei Liu received the BE degree in 2020 from the Department of Electronic Information Engineering, Nanjing University of Posts and Telecommunications, China, where he is currently working toward the master's degree with the Department of Computer Science and Technology. His research focuses on parallel and distributed computing.

Xiong Fu received the BS and PhD degrees in computer science from the University of Science and Technology, Hefei, in 2002 and 2007, respectively. He is currently a professor with the School of Computer Science, Nanjing University of Posts and Telecommunications. His research interests include parallel and distributed computing, and cloud computing.

Fu Xiao received the PhD degree in computer science and technology from the Nanjing University of Science and Technology, Nanjing, China, in 2007. He is currently a professor and PhD supervisor with the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, China. His research papers have been published in many prestigious conferences and journals, such as IEEE INFOCOM, IEEE ICC, IEEE IPCCC, IEEE/ACM ToN, IEEE JSAC, IEEE TMC, ACM TECS, and IEEE TVT. His research interests mainly include Internet of Things and mobile computing. He is a member of the IEEE Computer Society and the Association for Computing Machinery.

Chen Tian received the BS, MS, and PhD degrees from the Department of Electronics and Information Engineering, Huazhong University of Science and Technology, China, in 2000, 2003, and 2008, respectively. He is currently an associate professor with the State Key Laboratory for Novel Software Technology, Nanjing University, China. He was an associate professor with the School of Electronics Information and Communications, Huazhong University of Science and Technology, China. From 2012 to 2013, he was a postdoctoral researcher with the Department of Computer Science, Yale University. His research interests include data center networks, network function virtualization, distributed systems, Internet streaming, and urban computing.