APPENDIX A

A sample lock-free linked list used by DHASH

DHASH is modular, and its buckets can be implemented by using existing wait-free or lock-free set algorithms that implement the API shown in Algorithm 1. This appendix takes Michael's classic linked list [1] as an example and illustrates the modifications to a lock-free set algorithm before it can be integrated into DHASH. The pseudocode for the resulting algorithm is presented in Algorithm 7, with all modifications being highlighted with comments in bold fonts (i.e., **U1**, **U2**, etc.). Except for these modifications, Algorithm 7 is the same as Michael's original algorithm. Note that for ease of presentation, the *cas* primitive in this appendix returns *True* when it succeeds; otherwise, it returns *False*.

(1) The RCU mechanism [2] is used for memory reclamation. The memory reclamation mechanism presented in Michael's original algorithm (hazard pointers) leads to performance penalties due to the memory fence instructions in traversing the list. Therefore, in DHASH, we chose RCU. Moreover, when the RCU mechanism is properly applied, no ABA issue [3] can arise, such that the *tag* field in each node can be saved [1]. Specifically, in Algorithm 7, before operations holding references to a node have completed, the RCU mechanism prevents other concurrent delete operations from reclaiming (and then reusing) the node (lines 151 and 178). Moreover, the use of *call_rcu()* prevents delete operations from being blocked by prior unfinished lookup operations [2].

(2) Logically removed nodes are allowed to be inserted into the list. Recall that while the function *ht_rebuild()* is distributing a node α, a concurrent delete operation can find α via the global pointer *rebuild_cur* and delete α by setting the *LOGIC_RM* bit of the next field of α. One approach to solving this interaction is letting the function *list_insert()* check the *LOGIC_RM* bit when inserting α into the new hash table by using an atomic *double-compare-single-swap (dcss)* primitive (discussed below). That is, *list_insert()* inserts a node into the hash table, only if the node's *LOGIC_RM* bit has not been set. To avoid the overuse of the *dcss* primitive, however, we adopt another approach that is optimistic and lightweight. Specifically, *list_insert()* first inserts α into the list, and then pro-actively removes α from the list if its *LOGIC_RM* bit is found to be already set. This approach works because once the *LOGIC_RM* bit is set, it remains, and subsequent invocations of the function *list_find()* will help remove α out of the list, rather than returning α as the search result (discussed in detail in Section 4.5).

To achieve this, when setting the *next* field of α to point to its successor node (line 124, i.e., **U1**), we only change its *ptr* field, but save its *flag* field (line 118). Since concurrent delete operations may be logically deleting α, the helper function *set_next_pointer()* uses a loop that repeatedly sets the *next* field by using a *CAS* instruction. Since once the *LOGIC_RM* bit is set, it remains, the *cas* instruction fails at most once.

One consequence of allowing the insertion of logically removed nodes is that it can cause a concurrent *list_find()* operation, which is required to first help physically remove the node, to start over. Even though the *list_find()* operation

Algorithm 7: Lock-free, ordered linked list for DHASH.

```
109 struct node {long key; <node *ptr, flag> next};
110 struct list {node *head};
111 struct snapshot {node **prev, *cur, *next};
112 #define logically_removed(htnp) (htnp->next & LOGIC_RM)
113 set_snapshot(snapshot *ss, node **prev, *cur, *next) {
114     ss->prev := prev; ss->cur := cur; ss->next := next;
115 }
116 set_next_pointer(node *htnp, <new_ptr, new_flag>) {
117     do { <old_ptr, old_flag> := htnp->next;
118     } while (cas(&htnp->next, <old_ptr, old_flag>, <new_ptr,
          old_flag>));
119 }
120 list_insert(list *list, node *htnp) {
        Local variables : snapshot ss;
121     while (true) {
122         if (list_find(list, htnp->key, &ss) = SUCCESS)
123             return -EEXIST;
124         set_next_pointer(htnp, ss.cur);                    /* U1 */
125         if (cas(ss.prev, <ss.cur, 0>, <htnp, 0>)) {
126             if (logically_removed(htnp))                   /* U2 */
127                 list_find(list, htnp->key, &ss);           /* U2 */
128             return SUCCESS;
129         }
130     }
131 }
132 list_insert_dcss(ht **ht_new, *old1, list *list, node *htnp) {
        Local variables : snapshot ss; int ret;
133     while (true) {
134         if (list_find(list, htnp->key, &ss) = SUCCESS)
135             return -EEXIST;
136         htnp->next := <ss.cur, 0>;
137         ret := dcss(ht_new, old1, ss.prev, <ss.cur, 0>, <htnp, 0>);   /* U3
              */
138         if (ret = SUCCESS) return SUCCESS;
139         else if (ret = -EADDR1) return -EADDR1;            /* U4 */
140     }
141 }
142 list_delete(list *list, long key, long flag) {
        Local variables : snapshot ss; node *cur, *next;
143     while (true) {
144         if (list_find(list, key, &ss) != SUCCESS)
145             return -ENOENT;
146         cur := ss.cur; next := ss.next;
147         if (!cas(&cur->next, <next.ptr, 0>, <next.ptr, flag>))   /* U5 */
148             continue;
149         if (cas(ss.prev, <cur.ptr, 0>, <next.ptr, 0>))
150             if(logically_removed(cur))                     /* U6 */
151                 call_rcu(cur, free_node);
152         else list_find(list, key, &ss);
153         return SUCCESS;
154     }
155 }
156 list_find(list *list, long key, snapshot *ss) {
        Local variables : node **prev; node *cur_ptr, *next_ptr; long ckey;
157     try_again:
158     prev := &list->head;
159     <cur_ptr, prev_flag> := *prev;
160     while (true) {
161         if (cur_ptr = NULL) {
162             set_snapshot(ss, prev, NULL, NULL);
163             return -ENOENT;
164         }
165         <next_ptr, cur_flag> := cur_ptr->next;
166         ckey := cur_ptr->key;
167         if (*prev != <cur_ptr, 0>) goto try_again (line 157);
168         if (!cur_flag) {
169             if (ckey ≥ key) {
170                 set_snapshot(ss, prev, cur_ptr, next_ptr);
171                 return ckey = key ? SUCCESS : -ENOENT;
172             }
173             prev := &cur_ptr->next;
174         }
175         else { /* LOGIC_RM/IN_HAZARD has been set.          */
176             if (cas(prev, <cur_ptr,0>, <next_ptr,0>))
177                 if (logically_removed(cur_ptr))            /* U7 */
178                     call_rcu(cur_ptr, free_node);
179             else goto try_again (line 157);
180         }
181         <cur_ptr, prev_flag> := <next_ptr, cur_flag>;
182     }
183 }
```

is still non-blocking because the number of nodes inserted by *ht_rebuild()* is limited, this can cause an increased tail latency. To address this issue, after successfully inserting the node α into the list, the function *list_insert()* pro-actively checks whether α has been logically removed, and if so, invokes *list_find()* to physically remove α from the list (lines 126 - 127, i.e., **U2**). This guarantees that the number of deleted nodes not yet removed never exceeds the maximum number of concurrent threads operating on the list.

(3) A variant of *list_insert()* that supports the double-compare-single-swap (dcss) primitive is provided. Recall that DHASH stops inserting nodes into the old hash table once rebuild operations are in progress. Instead, it inserts nodes into the new hash table, to avoid inserting duplicate nodes (discussed in detail in Section 4.6).

To that end, we implement the function *list_insert_dcss()*, a variant of *list_insert()*, that allows us to perform an insert operation only when the *ht_new* field is equal to *NULL*, which implies that no rebuild operations are in progress. That is, the function *list_insert_dcss()* complies with the following sequential specification:

> **Definition 1**: The function *list_insert_dcss()* returns *-EADDR1* if rebuild operations are in progress (line 139, i.e., **U4**); otherwise, it returns *SUCCESS* if the node is successfully inserted into the list, and *-EEXIST* if such a node already existed in the list.

Specifically, the function *list_insert_dcss()* is similar to *list_insert()*, except that the linearization point of a successful *list_insert()* operation is replaced with Harris et al.'s lock-free double-compare-and-single-swap (*dcss*) primitive [4] (line 137, i.e., **U3**). The *dcss* primitive, which is implemented from normal *cas* instructions and a helping mechanism, 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.

(4) The function *list_delete()* takes a third parameter *flag*, which specifies the bit to be set in logical deletion (line 147, i.e., **U5**). This argument is either *LOGIC_RM* or *IN_HAZARD*. Correspondingly, when a node has been physically removed out of the list, *list_delete()* first checks its *flag* field before reclaiming the node (line 150, i.e., **U6**). If *flag* is set to *LOGIC_RM*, the node memory is reclaimed. Otherwise, the node is in its hazard period (i.e., a rebuild operation is referencing to this node via the global pointer *rebuild_cur*, and will insert this node into the new hash table), such that the node memory will not be reclaimed by *list_delete()*. Similarly, since the function *list_find()* may help concurrent delete operations to physically remove nodes from the list, it needs to first check that the node has been logically-removed, before reclaiming the node memory (line 177, i.e., **U7**).

(5) Wrapper functions are required when accessing the next fields of list nodes. One consequence of using the *dcss* primitive is that the value stored in the next field of every node is instead a *dcss* descriptor [4], which encapsulates the original value by shifting it to the left by one bit. Therefore, to read (resp. store) the original value from (resp. to) the *next* field, we need to use a pair of wrappers (*desc_2_ptr()* and *ptr_2_desc()*) which first perform a corresponding shift operation, and, in some rare cases, help a *dcss* operation [4]. The two wrappers are lightweight (please refer to the full paper for optimizations). The use of them, however, requires some programmer effort. Take the pseudocode shown in Algorithm 7 as the example, the wrappers are used on lines 117, 118, 124, 125, 126, 136, 137, 147, 149, 159, 165, 167, and 176 (13 lines of code in total). For ease of presentation, we omit them in the pseudocode, and refer the readers to the source code of DHASH, which will be open-sourced.

Correctness

We provide only informal proof that Algorithm 7 has the desired properties of a non-blocking linked list to be integrated into DHASH, with the following lemmas indicating the proof roadmap.

Lemma 7. *The functions* list_find(), list_delete(), *and* list_insert(), *when invoked by the regular hash table operations (*ht_lookup(), ht_delete(), *and* ht_insert()*), are exactly the same as Michael's original algorithm [1], and hence are linearizable to the sequential specification of standard set object.*

Proof. The function *ht_delete()* invokes *list_delete()* on lines 70 and 85, passing a third argument *LOGIC_RM*. By inspecting the pseudocode for *list_delete()*, it is easy to see that in this case, *list_delete()* performs the same as Michael's original algorithm. Similarly, the function *ht_insert()* invokes *list_insert()* on line 103. Since the node to be inserted into the list is a newly-allocated one, it is guaranteed that its *LOGIC_RM* bit has never been set. Therefore, *list_insert()* performs the same as the original algorithm. Moreover, *list_find()*, which is invoked by *ht_lookup()* on lines 54 and 64, may help the rebuild operation to physically remove a limited number of nodes in hazard period out of the list (line 176). Except that, *list_find()* performs the same as the original algorithm. □

Lemma 8. *The function* list_insert_dcss() *is linearizable and complies with the specification of the insert operation defined in Definition 1.*

Proof. Similar to the function *list_insert()*, *list_insert_dcss()* consists of a loop that repeatedly performs a lookup operation and the *dcss* instruction until either *dcss* succeeds (line 138), or a duplicate node is found (line 135). The only difference is that *list_insert_dcss()* can fail because a rebuild operation has started (line 139). In this case, it returns an error message *-EADDR1*, indicating that a rebuild operation has started and *ht_insert()* needs to start over. □

Lemma 9. *The functions* list_delete() *and* list_insert(), *when invoked by rebuild operations, can remove (resp. insert) a node from (resp. into) the list, without changing the abstract state of* DHASH.

Proof. Recall that the abstract state of DHASH can be defined as the node set H, and when rebuild operations are in progress, H is defined as 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 (discussed in detail in Section 5).

The function *ht_rebuild()* invokes *list_delete()* on line 33, passing a third argument *IN_HAZARD*. By inspecting the code, it is easy to see that in this case, *list_delete()* atomically sets the *IN_HAZARD* bit of the node (line 147), and then

the node will be removed from the list (line 149 or 176). On the other hand, Lemma 1 guarantees that concurrent lookup operations can always find the node via the global pointer *rebuild_cur*. That is, `H` still holds the node.

The function *ht_rebuild()* invokes *list_insert()* on line 37, and there are two cases.

(1) The node to be inserted may have been logically removed by a concurrent delete operation that found the node via the global pointer *rebuild_cur* (line 79). In this case, the delete operation has linearized, and the node has been removed from `H` (discussed in Section 4.5). Note that once a node's *LOGIC_RM* bit is set, it remains, and no concurrent lookup operations can return this node as the search result. For example, *list_insert()* keeps the *LOGIC_RM* bit while inserting this node into the list (line 124). Moreover, after the node has been successfully inserted into the list, *list_insert()* proactively removes the node by invoking *list_find()* (line 127).

(2) Otherwise, Lemma 1 guarantees that the global pointer *rebuild_cur* has pointed to the node before inserting it into the list. That is, this node has been in `H`.

Overall, when *list_insert()* is invoked by *ht_rebuild()*, the node set `H` and the abstract state of DHASH do not change. □

REFERENCES

[1] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *SPAA*, 2002.

[2] P. McKenney, "Introduction to RCU," http://www.rdrop.com/~paulmck/RCU/, 2020, [Online; accessed 9-May-2019].

[3] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[4] T. Harris, K. Fraser, and I. Pratt, "A practical multi-word compare-and-swap operation," in *DISC*, 2002.