Technical Report

Designing ASCY-compliant Concurrent Search Data Structures

A hash table and a binary search tree

Tudor David Rachid Guerroui Che Tong Vasileios Trigonakis *
Distributed Programming Lab (LPD), EPFL
name.surname@epfl.ch

Code available at:;
https://github.com/LPD-EPFL/CLHT and https://github.com/LPD-EPFL/ASCYLIB

*Authors appear in alphabetical order.

Abstract

This report details the design of two new concurrent data structures, a hash table, called CLHT, and a binary search tree (BST), called BST-TK. Both designs are based on *asynchronized concurrency (ASCY)*, a paradigm consisting of four complementary programming patterns. ASCY calls for the design of concurrent search data structures to resemble that of their sequential counterparts. CLHT (cache-line hash table) uses cache-line-sized buckets and performs in-place updates. As a cache-line block is the granularity of the cache-coherence protocols, CLHT ensures that most operations are completed with *at most* one cache-line transfer. BST-TK reduces the number of cache-line transfers by acquiring less locks than existing BSTs.

1 Introduction

A *search data structure* consists of a set of elements and an interface for accessing and manipulating these elements. The three main operations of this interface are a *search* operation and two update operations (one to *insert* and one to *delete* an element), as shown in Figure 1. Search data structures are said to be *concurrent* when they are shared by several processes. Concurrent search data structures (CSDSs) are commonplace in today's software systems. For instance, concurrent hash tables are crucial in the Linux kernel [10] and in Memcached [12], while skip lists are the backbone of key-value stores such as RocksDB [8]. As the tendency is to place more and more workloads in the main memory of multi-core machines, the need for CSDSs that effectively accommodate the sharing of data is increasing.

Nevertheless, devising CSDSs that scale and leverage the underlying number of cores is challenging [1, 2, 4, 9, 13]. Even the implementation of a specialized CSDS that would scale on a specific platform, with a specific performance metric in mind, is a daunting task. Optimizations that are considered effective on a given architecture might not be revealed as such on another [1, 6]. For example, NUMA-aware techniques provide no benefits on uniform architectures [6]. Similarly, if a CSDS is optimized for a specific type of workload, slightly different workloads can instantly cause a bottleneck. For instance, read-copy update (RCU) [11] is extensively used for designing CSDSs that are suitable for read-dominated workloads. However, it could be argued that this is achieved at the expense of scalability in the presence of updates.

Motivated by the aforementioned issues, we recently conducted a thorough evaluation and analysis of CSDSs on modern hardware [5]. This work resulted in *asynchronized concurrency (ASCY)*, a paradigm consisting of four complementary programming patterns. ASCY calls for the design of CSDSs to resemble that of their sequential counterparts. The four patterns are the following:

ASCY$_1$: The search operation should not involve any waiting, retries, or stores.

ASCY$_2$: The parse phase of an update operation should not involve any stores other than for cleaning-up purposes and should not involve any waiting, or retries.

ASCY$_3$: An update operation whose parse is unsuccessful (i.e., the element not found in case of a remove, the element already present in case of an insert) should not perform any stores, besides those used for cleaning-up in the parse phase.

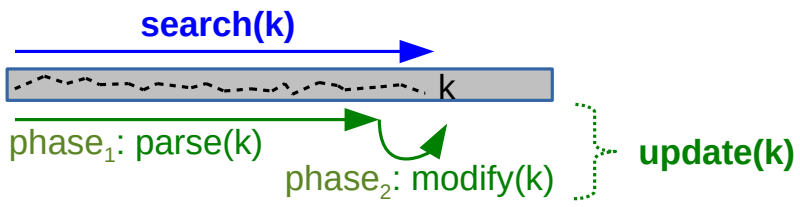

Figure 1: Search data structure interface. Updates have two phases: a parse phase, followed by a modification phase.

ASCY$_4$: The number and region of memory stores in a successful update should be close to those of a standard sequential implementation.

In addition, we showed how (i) the four rules can be simultaneously applied on existing CSDSs, and (ii) when they are applied they lead to designs that are *portably scalable* (i.e., they scale across hardware platforms, various workloads, and for different performance metrics, such as throughput and energy efficiency). Naturally, an interesting question to ask is whether ASCY can help in the design of new CSDS algorithms.

This report shows that ASCY can indeed help in the design of very efficient, novel concurrent data structures. In short, we detail the design of a concurrent hash table (CLHT) and a concurrent binary search tree (BST-TK). Both algorithms follow ASCY and outperform existing state of the art CSDSs.

CLHT takes its name from *cache-line hash table*, as it tries to put one bucket per cache line. Doing so, CLHT, in the common case, requires *at most* one cache-line transfer to complete an operation. The key ideas behind CLHT are (i) in-place updates of key/value pairs, and (ii) parsing the bucket with a snapshot of key/value pairs. CLHT comes in two variants, one lock-based and one lock-free.

BST-TK stands for *BST Ticket*, as it cleverly uses ticket locks both for locking and for keeping track of version numbers of nodes. In short, BST-TK parses the tree in a wait-free manner, thus search operations are wait-free. Insertions and removals parse the tree and then lock one and two nodes respectively. Parsing involves keeping track of the version number of the nodes, so that once the node is locked, the version can be validated in order to avoid concurrent conflicting modifications. We modify the interface of ticket locks so that the lock acquisition involves the version number of the node. Accordingly, we are able to perform locking and validate the version in a shingle step.

2 CLHT, Cache-Line Hash Table

The cache-line transfers triggered by the cache-coherence protocols are one of the largest impediments to software scalability on modern multi-cores [6]. Accordingly, concurrent software should aim at minimizing the amount of generated coherence traffic. This result can be achieved by reducing the number of cache lines that are being written[1]. The four ASCY patterns steer CSDSs towards precisely this direction by bringing them as close to their sequential counterparts as possible.

CLHT aims at reducing the number of cache-line transfers to the absolute minimum. Intuitively, when an update operation is completed (e.g., a new key/value pair is inserted in the hash table), at least one write on shared state has to be performed. Additionally, the granularity of coherence is one cache line, that is 64 bytes on most modern multi-cores. CLHT tries to complete operations with *at most* one cache-line transfer by using cache lines as buckets:

```
1  #define ENTRIES_PER_BUCKET 3
2
3  typedef uintptr_t clht_key_t;
4  typedef volatile uintptr_t clht_val_t;
5
6  struct bucket
7  {
8    uint64_t concurrency; // used for synchronization
9    clht_key_t key[ENTRIES_PER_BUCKET];
10   clht_val_t val[ENTRIES_PER_BUCKET];
11   struct bucket* next; // used to link buckets
12 }
```

A bucket consists of three key/value pairs (i.e., six words) and two words, one for linking buckets and one for implementing synchronization (e.g., a lock). Based on this bucket structure, it is straightforward to design a lock-based hash table that protects all three operations (i.e., search, insert, remove) with a lock. However, ASCY$_1$ explicitly suggests that search operations should not perform any stores. Consequently, the search operation of CLHT needs to parse the keys of the buckets and return without any synchronization. To achieve this, parsing the bucket must do more than just comparing the given key to the bucket's keys: an atomic snapshot of each key/value pair is taken. The atomic snapshot

[1]Recall that coherence traffic is mainly generated by a thread that needs to write on some data that are cached on a remote core.

```
1  /* Retrieve a key-value entry from CLHT-LB */
2  clht_val_t
3  clht_search(clht_hashtable_t* hashtable, clht_addr_t key)
4  {
5    size_t bin = clht_hash(hashtable, key);
6    volatile bucket_t* bucket = hashtable->table + bin;
7
8    uint32_t j;
9    do
10     {
11       for (j = 0; j < ENTRIES_PER_BUCKET; j++)
12         {
13           clht_val_t val = bucket->val[j];
14           if (bucket->key[j] == key)
15             {
16               if (bucket->val[j] == val)
17                 {
18                   return val;
19                 }
20               else
21                 {
22                   return 0;
23                 }
24             }
25         }
26
27       bucket = bucket->next;
28     }
29   while (bucket != NULL);
30   return 0;
31 }
```

Figure 2: Search operation of CLHT-LB.

guarantees that if a search finds the target key, the value that it will return corresponds to that key and not to a concurrent modification. The atomic snapshot can be easily taken with:

```
clht_val_t val = bucket->val[i];
if (bucket->key[i] == key && bucket->val[i] == val)
  {
    //we have an atomic snapshot of key[i] and val[i]
  }
```

The only requirement to be able to use the code snippet above is that the same value cannot be reused by a concurrent operation throughput the lifespan of the current operation. Otherwise, the current operation might read `val`, compare the `key`, and then return the `val` that has been removed and re-inserted by a concurrent operation. In CLHT, the aforementioned requirement is provided by the garbage collection of the SSMEM memory allocator [5].

Based on the aforementioned ideas, we develop two variants of CLHT, lock-based (CLHT-LB) and lock-free (CLHT-LF).

2.1 CLHT-LB, the lock-based variant of CLHT

CLHT-LB uses the `concurrency` word of a bucket as a lock. The design and implementation is quite straightforward.

Search operation. Figure 2 contains the code of the search operation. Lines 5-6 find the corresponding bucket for the given key. Lines 9-29 parse the bucket using atomic snapshots for each key/value pair. If the end of the bucket is reached, the operation tries to follow any linked buckets (Lines 27-29).

Insert operation. Figure 3 contains the code of the insert operation. Lines 8-11 implement read-only unsuccessful insertions ($ASCY_3$). The `bucket_exists` function is not shown, but is almost identical to the `clht_search` operation. Lines 23-35 traverse the bucket and either return `false` (if the key is found), or keep track for a potential empty spot for the insertion. In Lines 37-53, if there is no next bucket (i.e., there is no reason to keep searching), the new key/value pair is inserted, maybe after expanding the bucket in case there is no space left in the existing ones.

```
/* Insert a key-value entry into CLHT-LB. */
int
clht_insert(clht_hashtable_t* hashtable, clht_addr_t key, clht_val_t val)
{
  size_t bin = clht_hash(hashtable, key);
  bucket_t* bucket = hashtable->table + bin;

  if (bucket_exists(bucket, key)) //implement read-only fail (ASCY3)
    {
      return false;
    }

  clht_lock_t* lock = &bucket->lock;

  clht_addr_t* empty = NULL;
  clht_val_t* empty_v = NULL;

  uint32_t j;

  LOCK_ACQ(lock);
  do
    {
      for (j = 0; j < ENTRIES_PER_BUCKET; j++)
        {
          if (bucket->key[j] == key)
            {
              LOCK_RLS(lock);
              return false;
            }
          else if (empty == NULL && bucket->key[j] == 0)
            {
              empty = &bucket->key[j];
              empty_v = &bucket->val[j];
            }
        }

      if (bucket->next == NULL)
        {
          if (empty == NULL)
            {
              bucket->next = clht_bucket_create();
              bucket->next->key[0] = key;
              bucket->next->val[0] = val;
            }
          else
            {
              *empty_v = val;
              *empty = key;
            }

          LOCK_RLS(lock);
          return true;
        }

      bucket = bucket->next;
    } while (true);
}
```

Figure 3: Insert operation of CLHT-LB.

```
1  /* Remove a key-value entry from CLHT. */
2  clht_val_t
3  clht_remove(clht_hashtable_t* hashtable, clht_addr_t key)
4  {
5    size_t bin = clht_hash(hashtable, key);
6    bucket_t* bucket = hashtable->table + bin;
7
8    if (!bucket_exists(bucket, key))
9      {
10       return false;
11     }
12
13   clht_lock_t* lock = &bucket->lock;
14   uint32_t j;
15
16   LOCK_ACQ(lock);
17   do
18     {
19       for (j = 0; j < ENTRIES_PER_BUCKET; j++)
20         {
21           if (bucket->key[j] == key)
22             {
23               clht_val_t val = bucket->val[j];
24               bucket->key[j] = 0;
25               LOCK_RLS(lock);
26               return val;
27             }
28         }
29       bucket = bucket->next;
30     } while (bucket != NULL);
31   LOCK_RLS(lock);
32   return false;
33 }
```

Figure 4: Remove operation of CLHT-LB.

Remove operation. Figure 4 contains the code of the remove operation. The code for removals is pretty straightforward. Lines 8-11 implement read-only unsuccessful removals ($ASCY_3$). Lines 17-30 traverse the key/value pairs and if the key is found, it is removed from the hash table.

2.1.1 Correctness sketch

Search operations access the key/value pairs with a snapshot, thus, when the value is returned in Line 18 of Figure 2, it is certain that it is the result of a correct key/value pair insertion. In particular, we can take a key/value snapshot because (i) the values that can be seen during the lifespan of an operation are unique[2] and (ii) insertions first write the value and then the key when inserting a new key/value pair.

If the validation of the value fails (Lines 20-23), the operation can return 0. Since the key was found, but the snapshot fails, we know that there is at least one concurrent to this search removal (otherwise, without a concurrent removal the snapshot cannot fail, because an insertion first writes the value and then the key). We can simply select the linearization point of the search to be before the linearization point of the concurrent removal and before any possible insertion of the same key.

Finally, following a linked bucket does not change the game in terms of correctness. Since word-sized writes to the memory are atomic, when the search operation loads the `bucket->next` pointer, it will either find a non-`NULL` next pointer or not. Even if there is a concurrent insertion of the target key, the linearization point of search can be placed either before or after the insertion, depending on whether it observes the next pointer.

Insert operations might return 0 without ever acquiring the lock (Line 10 of Figure 3). Doing so is correct, because it is equivalent to performing a search operation that did find the key in the bucket. The actual insertions are protected by the lock, so their correctness is obvious as no concurrent updates are allowed.

[2]It is impossible to see value A twice because it was removed and then reused. This is guaranteed by the memory allocator which supports garbage collection.

Removals are correct based on the exact same reasoning as insertions.

2.1.2 Liveness

Search operations are clearly wait-free if we assume that buckets do not become infinitely long due to linking. In theory, this assumption is valid because we have a limited key space: keys are 64-bits long. In practice, the length of the buckets is kept short to optimize for performance (see "Resizing" below). Update operations are blocking as they rely on locks.

2.1.3 Resizing

CLHT-LB supports resizing. In Figure 3, a insertion that does not find an empty spot in the bucket simply expands the last bucket. Nevertheless, these expansions can make the buckets pretty lengthy, which is suboptimal in terms of performance. Accordingly, CLHT-LB keeps track of the number of total bucket expansions and when it is above a threshold, an actual hash-table resize is triggered.

The implementation of resizing is quite simple: a single thread traverses the buckets, acquires the locks and copies the contents one bucket after the other. Traversing the hash table is quite fast, because the main hash table is an array of buckets (i.e., accessing the data is sequential). Notice that concurrent search operations are unaware of the resize operation and can complete normally.

We also implement helping with multiple threads: when a thread finds a lock occupied due to a resize, it can help with resizing. However, our evaluation shows that helping is beneficial only with very large hash tables. On smaller hash tables, the cost of synchronizing the helper threads outweighs the performance benefits (recall that sequential memory accesses are extremely fast on modern hardware, hence a single thread can resize the hash table very efficiently).

2.1.4 Flavors

We implement various variants of CLHT-LB with different characteristics, mainly when it comes to how full buckets are handled. The most important "flavors" of CLHT-LB are:

- A variant that does not include `bucket->next` pointers. Instead, when a bucket is full, a hash-table resize is immediately triggered. This variant has the benefit of keeping the buckets short, but might result in a low-occupancy hash table.
- A variant where the buckets are linked to their next buckets (b_0 to b_1, b_1 to b_2, ...), so that if there is no space in a bucket, the next one is used. If the hash table is too full, it is resized. This variant can achieve very high occupancy of the hash table.
- A variant where remove operations do not acquire the lock, but perform the removal using a compare-and-swap on the key. This variant can give slightly better performance than the default one, but makes hash-table resizing more complicated.

2.2 CLHT-LF, the lock-free variant of CLHT

Synchronizing updates in CLHT-LB is straightforward: when a thread wants to perform a modification, it grabs the lock and atomically performs the changes. However, to create a lock-free version of CLHT, namely CLHT-LF, we need to discard those per-bucket locks. Without locks, the coordination of updates becomes quite complex.

A first idea could be to use compare-and-swap (CAS) for modifying the keys of the bucket. For instance, an insertion could CAS the target spot from empty to the required key. Clearly, this solution would cause atomicity problems for inserting both the key and the value. Additionally, if we do not explicitly coordinate concurrent insertions in the same bucket, we might end up with duplicate keys, inserted by two concurrent operations.

To solve this problem, we introduce the `clht_snapshot_t` structure:

```
typedef union
{
  volatile uint64_t snapshot;
  struct
  {
    uint32_t version;
    uint8_t map[ENTRIES_PER_BUCKET];
  };
} clht_snapshot_t;
```

`clht_snapshot_t` is 8-bytes long and can thus be loaded, stored, or CASed with a single operation (i.e., atomically). It contains a version number and a map. The version number is used to synchronize concurrent modifications, while the map is used to indicate whether the corresponding index in the bucket is valid, invalid, or if it is being inserted.

Figure 5 contains the interface of `clht_snapshot_t`. `snap_get_empty_index` find the index of the first, if any, empty spot in the snapshot structure. `snap_set_map` simply sets the value of the given index to the provided value. Finally, `snap_set_map_and_inc_version` also sets the value of the given index, but it further increments the version number by one. We use the interface of `clht_snapshot_t` in the design of CLHT-LF.

CLHT-LF's bucket uses the snapshot structure instead of CLHT-LB's lock:

```
typedef volatile struct ALIGNED(CACHE_LINE_SIZE) bucket_s
{
  union
  {
    volatile uint64_t snapshot;
    struct
    {
      uint32_t version;
      uint8_t map[ENTRIES_PER_BUCKET];
    };
  };
  clht_addr_t key[ENTRIES_PER_BUCKET];   clht_val_t val[ENTRIES_PER_BUCKET];
} bucket_t;
```

```
static inline int
snap_get_empty_index(uint64_t snap)
{
  clht_snapshot_t s = { .snapshot = snap };
  int i;
  for (i = 0; i < ENTRIES_PER_BUCKET; i++)
    {
      if (s.map[i] == MAP_INVLD)
        {
          return i;
        }
    }
  return -1;
}

static inline uint64_t
snap_set_map(uint64_t s, int index, int val)
{
  clht_snapshot_t s1 = { .snapshot = s };
  s1.map[index] = val;
  return s1.snapshot;
}

static inline uint64_t
snap_set_map_and_inc_version(uint64_t s, int index, int val)
{
  clht_snapshot_t s1 = { .snapshot =  s};
  s1.map[index] = val;
  s1.version++;
  return s1.snapshot;
}
```

Figure 5: Interface for manipulating `clht_snapshot_t` structures.

```
1  static inline clht_val_t
2  clht_bucket_search(bucket_t* bucket, clht_addr_t key)
3  {
4    int i;
5    for (i = 0; i < KEY_BUCKT; i++)
6      {
7        clht_val_t val = bucket->val[i];
8        if (bucket->map[i] == MAP_VALID)
9          {
10           if (bucket->key[i] == key)
11             {
12               if (bucket->val[i] == val)
13                 {
14                   return val;
15                 }
16               else
17                 {
18                   return 0;
19                 }
20             }
21         }
22     }
23   return 0;
24 }
25
26
27 /* Retrieve a key-value entry CLHT-LF. */
28 clht_val_t
29 clht_get(clht_hashtable_t* hashtable, clht_addr_t key)
30 {
31   size_t bin = clht_hash(hashtable, key);
32   bucket_t* bucket = hashtable->table + bin;
33
34   return clht_bucket_search(bucket, key);
35 }
```

Figure 6: Search operation of CLHT-LF.

Search operation. Figure 6 contains the code of the search operation. `clht_get` is based on the `clht_bucket_search` operation. This operation is similar to CLHT-LB's search operation. However, the snapshot includes not only the key and the value, but also the `clht_snapshot_t`'s map for the corresponding index. If the map does not contain the valid flag, the current key/value pair can be simply ignored.

Insert operation. Figure 7 contains the code of the insert operation. Line 12 contains the jump target in case the insertion must be restarted. The first thing a try does is to get the current snapshot of the bucket (Line 13). Lines 15-22 implement the read-only unsuccessful insertion ($ASCY_3$). The `empty_index` variable indicates the empty index that might have been "acquired" from a previous try. If there is such, then the empty is index is "released" in Line 19.

If the key is not found in the bucket (Lines 23+) and if there is no `empty_index` held from a previous try, then the snapshot is inspected for empty spots (Lines 24-26). If an empty spot is found, the `map[empty_index]` is set to `MAP_INSRT` to indicate that a new key/value pair is being inserted to this spot. If either finding an empty spot, or changing the snapshot with the updated value (Line 32) fails, the operation is restarted.

If `empty_index` is greater or equal to 0 in Line 24, then the snapshot is updated in Line 43, as there is no need to look for a new empty spot. In short, the `MAP_INSRT` intermediate value is similar to a lock, with the main difference that it does not prohibit the other threads to complete their operations on other spots in a bucket.

Finally, in Lines 46-47, the thread tries to set the `map[empty_index]` to valid and to increment the version number. If it fails, the operation is restarted, otherwise, the operation is completed.

```
/* Insert a key-value entry into CLHT-LF. */
int
clht_put(clht_t* h, clht_addr_t key, clht_val_t val)
{
  clht_hashtable_t* hashtable = h->ht;
  size_t bin = clht_hash(hashtable, key);
  bucket_t* bucket = hashtable->table + bin;

  int empty_index = -1;
  clht_snapshot_all_t s, s1;

 retry:
  s = bucket->snapshot;

  if (clht_bucket_search(bucket, key) != 0)
    {
      if (empty_index >= 0)
        {
          bucket->map[empty_index] = MAP_INVLD;
        }
      return false;
    }

  if (empty_index < 0)
    {
      empty_index = snap_get_empty_index(s);
      if (empty_index < 0)
        {
          goto retry;
        }
      s1 = snap_set_map(s, empty_index, MAP_INSRT);
      if (CAS_U64(&bucket->snapshot, s, s1) != s)
        {
          empty_index = -1;
          goto retry;
        }

      bucket->val[empty_index] = val;
      bucket->key[empty_index] = key;
    }
  else
    {
      s1 = snap_set_map(s, empty_index, MAP_INSRT);
    }

  clht_snapshot_all_t s2 = snap_set_map_and_inc_version(s1, empty_index, MAP_VALID);
  if (CAS_U64(&bucket->snapshot, s1, s2) != s1)
    {
      goto retry;
    }

  return true;
}
```

Figure 7: Insert operation of CLHT-LF.

```
1  /* Remove a key-value entry from CLHT-LF. */
2  clht_val_t
3  clht_remove(clht_t* h, clht_addr_t key)
4  {
5    clht_hashtable_t* hashtable = h->ht;
6    size_t bin = clht_hash(hashtable, key);
7    bucket_t* bucket = hashtable->table + bin;
8
9    clht_snapshot_t s;
10   int i;
11  retry:
12   s.snapshot = bucket->snapshot;
13
14   for (i = 0; i < KEY_BUCKT; i++)
15     {
16       if (bucket->key[i] == key && s.map[i] == MAP_VALID)
17         {
18           clht_val_t removed = bucket->val[i];
19
20           clht_snapshot_all_t s1 = snap_set_map(s.snapshot, i, MAP_INVLD);
21           if (CAS_U64(&bucket->snapshot, s.snapshot, s1) == s.snapshot)
22             {
23               return removed;
24             }
25           else
26             {
27               goto retry;
28             }
29         }
30     }
31   return 0;
32 }
```

Figure 8: Remove operation of CLHT-LF.

Remove operation. Figure 8 contains the code of the remove operation. The removal is much simpler than the insert operation. The bucket is traversed and if the key is found in a valid state, the value is kept (Line 18) and the thread tries to set the `map[i]` value to invalid (Lines 20-21). If the latter operation fails, the removal is restarted (Line 27).

2.2.1 Correctness sketch

The map fields of a `clht_snapshot_t` object take three distinct values: `MAP_VALID`, `MAP_INVLD`, and `MAP_INSRT`. The first represents a valid key/value pair, the second an invalid one, and the third one that is currently being inserted and is thus still invalid.

As with CLHT-LB, the search in CLHT-LF takes a snapshot of key/value pairs, plus the value of the map for this index. If the snapshot is taken (Line 14 of Figure 6), then we know that the value corresponds to the key and the map contains a valid flag for this pair. However, proving the correctness of returning 0 in Line 18 if the snapshot fails is more complicated. We know that at the point where the thread reads `map[i]` there is a valid key/value pair, otherwise the if statement of Line 8 would fail. The same applies for Line 10: the `key[i]` is the key that the operation is looking for. Consequently, we can recognize two reasons why the snapshot failed:

1. The valid value of `map[i]` corresponds to a different key. In this case, a concurrent removal has to remove the key by setting `map[i]` to invalid and a subsequent, but still concurrent, insertion must write the value of key that the search is looking for. Therefore, the insertion of key is concurrent to the current search, hence the search can be linearized before the insertion.

2. The valid value of `map[i]` corresponds to the correct key. In this case, the snapshot can only fail if there is again a concurrent removal with a subsequent insert (which changes the value). Therefore, the removal of key is concurrent to the search, hence the search can be linearized after the removal.

If an insertion finds the key in the bucket, then it returns (Lines 15-22 of Figure 7). This is trivially

correct, as it can be seen as a search operation that found the key. In addition, if the `empty_index` is greater or equal to zero, the value of `map[i]` is reverted to `MAP_INVLD`. This change in `map[i]` does not affect searches or removals, as, for those, `MAP_INSRT` is equivalent to `MAP_INVLD`.

Every retry of the operation starts by taking the current value of the `clht_snapshot_t` object in order to ensure that we avoid "problematic concurrency".

`empty_index` is a simple state diagram: if it is less than 0, then the current operation does not hold any spot in the bucket for its insertion, otherwise, it holds the spot that corresponds to the value of `empty_index`.

In the former case (Lines 24-40), the operation looks for an empty spot in Line 26 and if there is not any (i.e., the bucket is full with valid and being inserted spots), the operation is restarted (Line 29). If a spot is found, the thread tries with a CAS to update the `clht_snapshot_t` object so that it is visible to the other threads that an insertion is happening on this spot. If this CAS fails, the operation is restarted, after the `empty_index` is set to negative to show that it has not acquired any spots. This CAS might fail either because there was a change in the values of the map, or a version change. The former might be a false conflict, but the latter is necessary so that we disallow concurrent insertions of the same key on different spots of the same bucket. If the CAS succeeds, nobody can change this spot, thus the thread can safely write the value and the key to be inserted.

If the `empty_index` is greater or equal to zero in Line 24, we know that a previous try has successfully performed the steps of Lines 25-39, thus we only need to update the `clht_snapshot_t` object so that it gets then new version number and the new state of the other map fields.

Once the previous steps are completed, a new version of the `clht_snapshot_t` object is created with (i) an incremented version number, and (ii) `map[empty_index]` set to valid. In other words, we are trying to make the insertion visible to the other threads with a CAS in Line 47. Clearly, if there are many concurrent insertions, only one can succeed, because of the version number that is incremented. Accordingly, we avoid the case of concurrent insertions putting the same key in different spots of the bucket.

Finally, the correctness of the remove operation is trivial. A removal can only succeed if there is no completed concurrent insertion, otherwise the version number would have increased, making the CAS (Line 21 of Figure 8) fail. Even without concurrent insertions, concurrent removals might cause the current removal to restart. In any case, if the CAS fails, we ensure that the correct key/value pair is removed.

2.2.2 Liveness

We can easily see that the search operation is wait-free, as it always executes a finite number of steps since there is no way to be restarted. Updates are lock-free under certain assumptions: the number of crashed processes that is supported depends on the longest possible bucket (i.e., how many keys fall in the same bucket). A crashed process might render one of the spots of the bucket useless by leaving the `MAP_INSRT` value before crashing.

A removal can be restarted by either a concurrent successful update (which is OK for lock-freedom), or by an insertion preparing the insert (i.e., setting the value of a map entry to `MAP_INSRT`). However, we can have a finite number of the latter restarts before an insert operation completes: in the extreme case, all spots in the buckets will be set to `MAP_INSRT`. When this happens, the first insertion that will do CAS to increment the version will be successful.

```
update()
{
 retry:
  parse(); /* keeps track of the version numbers */
  if (!can_update())    /* ASCY3 */
    {
      return false;
    }
  lock(); /* 1 node for insert, 2 for remove */
  if (!validate_version())
    {
      goto retry;
    }
  apply_update();
  increase_version();
  unlock();
}
```

Figure 9: Steps of update operations in BST-TK.

3 BST-TK, Binary Search Tree Ticket

Existing lock-based binary search trees (BST) [3, 7] are (i) quite complex and (ii) acquire a large number of locks, more than we could consider as minimum. Accordingly, these designs do not follow the ASCY_4 pattern that suggests that the modification phase of an update (i.e., when the writes happen) should be close to the one of any standard sequential implementation.

To address the lack of an "ASCY-compliant" lock-based BST, we develop BST-TK. Intuitively, on any lock-based BST an update operation parses to the node to modify and, if possible, acquires a number of locks and performs the update. This is precisely how BST-TK proceeds.

More specifically, BST-TK is an external tree, where every internal (router) node is protected by a lock and contains a version number. The version numbers are used in order to be able to optimistically parse the tree and later detect concurrency. Update operations proceed as described in Figure 9.

We are able to simplify the design of BST-TK by making the simple observation that a ticket lock already contains a version field. Accordingly, we consolidate the version validation and increment, with locking and unlocking, respectively.

We do this by modifying the interface of the ticket lock in order to try to acquire a specific version of the lock (i.e., the one that the parsing phase has observed). If the lock acquisition fails, the version of the lock has been incremented by a concurrent update, hence the operation has to be restarted. We further optimize the tree by assigning two smaller (32-bits) ticket locks to each node, so that the left and the right pointers of the tree can be locked separately. Overall, BST-TK acquires one lock for successful insertions and two locks for successful removals.

Figure 10 contains the modified interface of the ticket lock. `tl_trylock_version` tries to acquire either the left or the right smaller ticket locks based on the given version (`tl_old`). `tl_trylock_version_both` tries to acquire both left and right locks at the same time (it is used by the remove operation). Finally, `tl_unlock` and `tl_revert` increment and decrement the version and the ticket values, respectively.

Accordingly, a BST-TK node is:

```
typedef struct node
{
  skey_t key;
  union
  {
    sval_t val;
    volatile uint64_t leaf;
  };
  volatile struct node* left;
  volatile struct node* right;
  volatile tl_t lock;
} node_t;
```

```c
typedef union tl32
{
  struct
  {
    volatile uint16_t version;
    volatile uint16_t ticket;
  };
  volatile uint32_t to_uint32;
} tl32_t;

typedef union tl
{
  tl32_t lr[2];
  uint64_t to_uint64;
} tl_t;

static inline int
tl_trylock_version(volatile tl_t* tl, volatile tl_t* tl_old, int right)
{
  uint16_t version = tl_old->lr[right].version;
  if (version != tl_old->lr[right].ticket)
    {
      return 0;
    }

  tl32_t tlo = { .version = version, .ticket = version };
  tl32_t tln = { .version = version, .ticket = (version + 1) };
  return CAS_U32(&tl->lr[right].to_uint32, tlo.to_uint32, tln.to_uint32) == tlo.to_uint32;
}

#define TLN_REMOVED  0x0000FFFF0000FFFF0000LL

static inline int
tl_trylock_version_both(volatile tl_t* tl, volatile tl_t* tl_old)
{
  uint16_t v0 = tl_old->lr[0].version;
  uint16_t v1 = tl_old->lr[1].version;
  if (v0 != tl_old->lr[0].ticket || v1 != tl_old->lr[1].ticket)
    {
      return 0;
    }

  tl_t tlo = { .to_uint64 = tl_old->to_uint64 };
  return CAS_U64(&tl->to_uint64, tlo.to_uint64, TLN_REMOVED) == tlo.to_uint64;
}


static inline void
tl_unlock(volatile tl_t* tl, int right)
{
  COMPILER_NO_REORDER(tl->lr[right].version++);
}

static inline void
tl_revert(volatile tl_t* tl, int right)
{
  COMPILER_NO_REORDER(tl->lr[right].ticket--);
}
```

Figure 10: The modified interface of a ticket lock.

```
1  sval_t
2  bst_tk_search(bst_tk_t* set, skey_t key)
3  {
4    node_t* curr = set->head;
5
6    while (!curr->leaf)
7      {
8        if (key < curr->key)
9          {
10           curr = (node_t*) curr->left;
11         }
12       else
13         {
14           curr = (node_t*) curr->right;
15         }
16     }
17
18   if (curr->key == key)
19     {
20       return curr->val;
21     }
22
23   return 0;
24 }
```

Figure 11: Search operation of BST-TK.

Search operation. Figure 11 contains the code of the search operation. The code could very well be the code for a standard sequential BST ($ASCY_1$ at its best). The operations traverses the tree until it reaches a leaf node. If that node is the one that the operation is looking for, then the value is returned.

Insert operation. Figure 12 contains the code of the insert operation. The parse phase (Lines 12-30) is identical to the search operation, with the addition that it keeps track of the predecessor node apart from the current one. Lines 32-35 implement the read-only unsuccessful insertions ($ASCY_3$). If the update is possible (Lines 36+), two new nodes are allocated, the one that holds the new key/value pair and one router node. Then, one lock is acquired, protecting either the left or the right pointer of the predecessor. If the locking succeeds, the operation proceeds, otherwise it is restarted.

Actually, some of these restarts can be avoided if we are willing to sacrifice complexity. In particular, if the version of the right or left pointer of the predecessor has changed due to an insertion, then instead of restarting, we could perform block-waiting and the proceed with the operation by traversing the new node(s).

```
1  int
2  bst_tk_insert(intset_t* set, skey_t key, sval_t val)
3  {
4    node_t* curr;
5    node_t* pred = NULL;
6    volatile uint64_t curr_ver = 0;
7    uint64_t pred_ver = 0, right = 0;
8
9   retry:
10   curr = set->head;
11
12   do
13     {
14       curr_ver = curr->lock.to_uint64;
15
16       pred = curr;
17       pred_ver = curr_ver;
18
19       if (key < curr->key)
20         {
21           right = 0;
22           curr = (node_t*) curr->left;
23         }
24       else
25         {
26           right = 1;
27           curr = (node_t*) curr->right;
28         }
29     }
30   while(!curr->leaf);
31
32   if (curr->key == key)
33     {
34       return 0;
35     }
36
37   node_t* nn = new_node(key, val, NULL, NULL, 0);
38   node_t* nr = new_node_no_init();
39
40   if ((!tl_trylock_version(&pred->lock, (volatile tl_t*) &pred_ver, right)))
41     {
42       ssmem_free(alloc, nn);
43       ssmem_free(alloc, nr);
44       goto retry;
45     }
46
47   if (key < curr->key)
48     {
49       nr->key = curr->key;
50       nr->left = nn;
51       nr->right = curr;
52     }
53   else
54     {
55       nr->key = key;
56       nr->left = curr;
57       nr->right = nn;
58     }
59
60   if (right)
61     {
62       pred->right = nr;
63     }
64   else
65     {
66       pred->left = nr;
67     }
68
69   tl_unlock(&pred->lock, right);
70
71   return 1;
72 }
```

Figure 12: Insert operation of BST-TK.

Remove operation. Figure 13 contains the code of the remove operation. Parsing for removals (Lines 11-31) keeps track of both the predecessor (`pred`) and the predecessor of the predecessor (`ppred`), because a removal affects both nodes. Lines 33-36 implement the read-only unsuccessful removals ($ASCY_3$). If the update is possible (Lines 37+), the appropriate, right or left, pointer of the `ppred` is locked, as well as both pointers of the `pred`. A removal needs more locks than an insertion, because it results in `pred` being completely removed from the tree. If both lock acquisitions are successful, then the actual removal is performed, otherwise the operation has to be restarted.

```
sval_t
bst_tk_remove(intset_t* set, skey_t key)
{
  node_t* curr, * pred = NULL, * ppred = NULL;
  volatile uint64_t curr_ver = 0;
  uint64_t pred_ver = 0, ppred_ver = 0, right = 0, pright = 0;

 retry:
  curr = set->head;

  do
    {
      curr_ver = curr->lock.to_uint64;
      ppred = pred;
      ppred_ver = pred_ver;
      pright = right;
      pred = curr;
      pred_ver = curr_ver;

      if (key < curr->key)
        {
          right = 0;
          curr = (node_t*) curr->left;
        }
      else
        {
          right = 1;
          curr = (node_t*) curr->right;
        }
    }
  while(!curr->leaf);

  if (curr->key != key)
    {
      return 0;
    }
  if ((!tl_trylock_version(&ppred->lock, (volatile tl_t*) &ppred_ver, pright)))
    {
      goto retry;
    }
  if ((!tl_trylock_version_both(&pred->lock, (volatile tl_t*) &pred_ver)))
    {
      tl_revert(&ppred->lock, pright);
      goto retry;
    }

  if (pright)
    {
      if (right)
        {
          ppred->right = pred->left;
        }
      else
        {
          ppred->right = pred->right;
        }
    }
  else
    {
      if (right)
        {
          ppred->left = pred->left;
        }
      else
        {
          ppred->left = pred->right;
        }
    }

  tl_unlock(&ppred->lock, pright);
  ssmem_free(alloc, curr);  ssmem_free(alloc, pred);
  return curr->val;
}
```

Figure 13: Remove operation of BST-TK.

3.1 Correctness proof

We rewrite the code of of BST-TK in a formatting more suitable for proving correctness.

```
1
2  Interface Vlock {
3    ▷ Lock one side of the ticket lock. It locks left/right lock when s is false/true.
4    bool LockOneSide(bool s, int ver);
5     ▷ Lock both sides of the ticket lock. It locks left/right lock when s is false/true.
6    bool LockBothSide(int ver);
7    int GetVer();
8    void Release(bool side);
9  };
10
11 type Node {
12   Node *right;
13   Node *left;
14   int key;
15   bool leaf;
16   Vlock lock;
17 };
18
19 Node *root;
20
21 INIT()  {
22   root = pointer to a new internal node with key −∞.
23   root → left = pointer to a new leaf node with key −∞.
24   root → right = pointer to a new leaf node with key +∞.
25 }
26
27 bool FIND(int k){
28   curr := root;
29   while curr points to an internal node  {
30       if  k < curr → key then curr := curr → left;
31       else  curr := curr → right;
32   }
33   if curr → key = k then return true;
34   return false;
35 }
36
37 bool INSERT(int k){
38   retry:
39   curr := root;
40   while !curr → leaf {
41       pred_ver := curr → lock.GetVer();
42       pred := curr;
43       if key ≤ curr → key then curr := curr → left;
44       else  curr := curr → right;
45   }
46   key := curr → key
47   if k = key then return false;
48   nn := pointer to a new leaf node with key k
49   nr := pointer to a new internal node with key min{key, k}
50   and child pointers curr and nr.
51   side := k > pred → key;
52    if pred → lock.LockOneSide(side, curr_ver) then goto retry;
53   ▷ Change one child pointer of pred to nr.
54   do_insert(pred,nr);
55   pred → Release(side);
56   return true;
57 }
58
59 bool REMOVE(Key k){
60 retry:
61   pred := root;
62   curr := root → right;
63   while !curr → leaf {
64     curr_ver = curr → lock.GetVer();
65     ppred := pred;
66     ppred_ver := pred_ver;
67     pred := curr;
68     pred_ver := curr_ver;
69     if k < curr → key then curr := curr → left;
70     else curr = curr → right;
71     }
72   if curr → key ≠ k then return false;
73   pside := k > ppred → key ;
```

```
74    if !ppred → lock.LockOneSide(pside, ppred_ver) then goto retry;
75    if !pred → lock.LockBothSide(pred_ver) then {
76      ppred → releaselock();
77      goto retry;
78  }
79  ▷ Change the pointer.
80    do_remove(ppred,pred,k);
81    ppred → releaselock(pside);
82    return true;
83  }
84
85  void do_insert(Node* p, Node* n){
86    ▷ Precondition: n points to a leaf node.
87    ▷ Precondition: p points to an internal node.
88    ▷ Precondition: Node pointed by n is a successor of node pointed by p.
89    if p → key ≤ n → key then
90      p → left := n;
91      else  p → right := n;
92
93  void do_remove(Node *pp, Node *p, int k){
94    ▷ Precondition: p and pp point to internal nodes.
95    ▷ Precondition: Node pointed by p is a successor of node pointed by pp.
96    ▷ Precondition: p has a leaf successor with key k.
97
98    if pp → key > k then {
99      if p → key > k then
100     pp → right := p → left;
101     else  pp → right := p → right;
102   }
103   else {
104     if p → key > k then
105     pp → left := p → left;
106     else  pp → left := p → right;
107   }
108 }
```

Listing 1: BST-TK rewritten for the correctness proof.

We have the following observations:

Lemma 1. *The pointer root never changes. The key field of any node never changes. The leaf field of any node never changes. Leaf nodes have no successors, Internal nodes always have two successors.*

Lemma 2. *Each call to do_remove and do_insert function satisfies preconditions.*

Proof. This can be easily deduced by the property of the ticket lock. For example, line 41 get the version of the lock, so we can conclude if line 52 successfully grasp the lock, there must not be any other thread modified the link from *pred* to *nr*. So if the locks are successfully grasped, the preconditions are automatically satisfied. □

We call a node is active after it is inserted to the data structure. Namely, after an insertion of line 54, the nodes pointed by *nn* and *nr* are inserted into the tree.

Lemma 3. *Any removed internal node is both side locked by some REMOVE operation and never released.*

Proof. Assume u is a removed internal node. Consider the execution step which this node is removed from the tree. It is easy to see the execution step must correspond to the execution of line 80. Because in the execution of line 54, only one child pointer of the node pointed by *pred* changes from *curr* to *nr*, but after the modification, the node pointed by *curr* is still on the subtree rooted at *pred*. So no node would be removed from the tree. However, in the execution of line 80, if the node pointed by *ppred* is not on the backbone, then the operation would obviously not remove any node from the backbone. So we have the following observation: if a computation step removes a backbone internal node, it is an computation step corresponding to line 80 and it removes the node pointed by *pred*. We can see the node is locked on both sides and the lock would not be released any more. □

Lemma 4. *If the left (or right) child field of any node u is modified by some thread T, the corresponding lock $u.lock.left$ (or $u.lock.right$) must have been acquired by the thread.*

Proof. It is easy to check before the execution of line 54 and line 80, the operation has to acquire the lock successfully. □

Lemma 5. *For an effectful INSERT operation, the last node pointed by pred locked by the operation is an internal backbone node. After grasping the lock, pred node has a child leaf node pointer field curr.*

Proof. From the Lemma 2, removed nodes are locked by the operation which does the removal. So the node pointed *pred* is on the backbone. The version number remained the same as the version number when *curr* advances to the child pointer of *pred*. So the node which *curr* points to is a leaf node, it is a child node of *pred*, after *pred* is locked. □

Lemma 6. *For effectful REMOVE operations, after grasping the two locks successfully, the node pointed by ppred is an internal backbone node with a child internal node pointed by pred. pred has a child leaf node which is pointed by curr.*

Proof. There is a state before acquisition of the locks such that *pred* is a child pointer of the node pointed by *ppred* , *curr* is a child pointer of the node pointed by *pred*. The version numbers are read before the pointer reference, so the success in lock acquisition implies that these links are not modified since that time. Also, from lemma 2, a removed node is locked permanently. So the node pointed by *ppred* is an internal backbone node after the lock acquisition. □

Lemma 7. *All the nodes reachable from root forms exactly an external binary search tree with BST ordering: if u is a node on the tree, v and w are the left and right children of u, then we have $v.key \leq u.key < w.key$.*

Proof. We prove this lemma by induction. The lemma obviously hold after initialization. We assume for a prefix of the execution α, the resulting configuration C satisfies above assumption. We consider the next computation step x. We want to prove the new configuration C' after the execution of x satisfies also the assumptions. We only need to consider the execution of line 54 and line 80. It is easy to verify execution of line 54 and line 80 preserves the tree invariant. Consider the execution of line 54. Since *do_insert* function satisfies its precondition, *nr* and *nn* points to two newly allocated nodes, *pred* change one of its successor to *nr*. From the implementation of *do_insert*, it is obvious that the tree invariant is preserved, because the node pointed by *pred* is on the tree from Lemma 4, then the modification is simply to replace the subtree of *nn* by a new subtree containing 3 nodes.

The execution of line 80 is similar. In configuration C, the node u pointed by *ppred* is on the backbone from Lemma 4, the nodes v, w pointed by *pred* and *curr* is the child and grandchild of the node pointed by u. It is easy to verify after the pointer modification, the invariants are still preserved. □

If we view the computation of the algorithm as an sequence of program states $\gamma_0, \gamma_1, \cdots \gamma_n$, then we can define the abstract state $Abs(\sigma)$ as the set of keys contained in all the leaf nodes reachable from root in state σ. Let $O_1, O_2, \cdots O_k$ be the subsequence of all effectful operations ordered by their linearizability points, i.e., the do_insert and do_remove functions. Assume $\sigma_0, \sigma_1, \cdots \sigma_k$ be the subsequence of states just after the execution of the linearizability points of the operations, we have the following lemma:

Lemma 8. *If O_i is an effectful INSERT(k) operation, then $k \notin Abs(\sigma_{i-1})$, and $k \in Abs(\sigma_i)$. If O_i is an REMOVE(k) operation, then $k \in Abs(\sigma_{i-1})$, and $k \notin Abs(\sigma_i)$.*

Proof. Lemma 4 and the precondition of do_insert guarantee that the do_insert operation inserts a new leaf node with key k to the tree at its linearizability point, other leaf nodes remains unchanged. Lemma 5 and the precondition of do_remove imply that $k \in Abs(\sigma_{i-1})$, and $k \notin Abs(\sigma_i)$, because a BST cannot have two leaf nodes with the same key. □

Lemma 9. *The BST-TK algorithm satisfies the assumption* RUA.

Proof. From Lemma 3, we know that it is obviously because all removed nodes are locked and never released. From Lemma 4, any thread cannot change the successor of removed nodes. So the algorithm satifies *RUA*. □

We add effectless operations to the argument. Effectless operations are linearized at non-fixed points. First, notice that all operations need to traverse the tree first, the traversal may restart several times. Assume the last traversal of an effectless operation visits nodes $v_1, v_2, \cdots v_n$. $v_1 = root$, v_n is a leaf node. We define the linearizability point of an effectless operation as the last state when v_{n-1} is not removed from the tree before *curr* goes through the pointer from v_{n-1} to v_n. This linearizability point is well defined, because of the Generalized Hindsight Lemma. Obviously, the temporal node path $v_1, v_2, \cdots v_n$ is a temporal backbone.

Lemma 10. *Let $O_1, O_2 \cdots O_k$ be the subsequence of all effectful operations ordered by their linearizability points. O is an effectless operation which is linearized at state σ between O_{i-1} and O_i with the linearizability point defined as above. Then : 1. If O is a FIND(k) operation, then O returns true if and only $k \in Abs(\sigma)$. 2. If O is an INSERT(k) operation, then $k \in Abs(\sigma)$. 3. If O is a REMOVE(k) operation, then $k \notin Abs(\sigma)$.*

Proof. The three cases are similar. They just claim that if a tree traversal looking for key k would end up with a leaf node with key k if and only if $k \in Abs(\sigma)$. If the traversal ends up with a leaf node v_n with key k, at state σ, v_{n-1} is a backbone node, v_n is a child of v_{n-1}. So $k \in Abs(\sigma)$. If the traversal ends up with a leaf node with $k' \notin k$. Obviously, if we change the key of the leaf node from k' to k, is also a valid BST, it satisfy all the constraints. So $k \notin Abs(\sigma)$. □

Lemma 11. *The algorithm is linearizable with the order defined by their linearizability points.*

Proof. Combining Lemma 8 and Lemma 10, the linearization order is given by the linearizability points. □

4 Conclusions

We presented a novel concurrent hash-table algorithm, namely CLHT, and a new lock-based binary search tree, called BST-TK. Both designs are based on asynchronized concurrency and show how with these four simple guidelines in mind we can design new, highly scalable concurrent search data structures.

References

[1] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. SOSP 2009.

[2] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. OSDI 2010.

[3] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A Practical Concurrent Binary Search Tree. PPoPP 2010.

[4] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 199–210. ACM, 2012.

[5] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. ASPLOS 2015.

[6] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. SOSP 2013.

[7] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical Concurrent Binary Search Trees via Logical Ordering. PPoPP 2014.

[8] Facebook. RocksDB. `http://rocksdb.org`.

[9] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. NSDI 2013.

[10] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. Scaling Dcache with RCU. *Linux Journal*, 2004(117), January 2004.

[11] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[12] Memcached. `http://www.memcached.org`.

[13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. NSDI 2013.