Technical Report

UCAM-CL-TR-639
ISSN 1476-2986

Number 639

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Non-blocking hashtables with open addressing

Chris Purcell, Tim Harris

September 2005

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Non-blocking Hashtables with Open Addressing

Chris Purcell Tim Harris

Abstract

We present the first non-blocking hashtable based on open addressing that provides the following benefits: it combines good cache locality, accessing a single cacheline if there are no collisions, with short straight-line code; it needs no storage overhead for pointers and memory allocator schemes, having instead an overhead of two words per bucket; it does not need to periodically reorganise or replicate the table; and it does not need garbage collection, even with arbitrary-sized keys. Open problems include resizing the table and replacing, rather than erasing, entries. The result is a highly-concurrent set algorithm that approaches or outperforms the best externally-chained implementations we tested, with fixed memory costs and no need to select or fine-tune a garbage collector or locking strategy.

1 Introduction

This paper presents a new design for non-blocking hashtables in which collisions are resolved by open addressing, i.e. probing through the other buckets of the table, rather than external chaining through linked lists.

The key idea is that rather than leaving tombstones to mark where deletions occur, we store per-bucket upper bounds on the number of other buckets that need to be consulted. This means that unlike the earlier designs we discuss in Section 2.2, ours supports a mixed workload of insertions and deletions without the need to periodically replicate the table's contents to clean out tombstones. Consequently, the table can operate without the need for dynamic storage management so long as its load factor remains acceptable.

Our design is split into three parts. Section 3.1 deals with maintaining the shared bounds associated with each bucket. The key difficulty here is ensuring that a bound remains correct when several entries are being inserted and removed at once. Section 3.2 builds on this to provide a hashtable. The main problem in doing so is guaranteeing non-blocking progress while ensuring that at most one instance of any key can be present in the table. In Section 3.3, we present a more complicated design allowing larger keys and a better progress guarantee, and in Section 3.4 we discuss open problems with the algorithm.

Section 4 evaluates the performance of our algorithm, compared to state-of-the-art designs based on external chaining. As with these, we rely only on the single-word atomic operations found on all modern processor families. Additionally, our algorithm has many properties that machines rely on for optimal performance: operations run independently, updating disjoint memory locations (*disjoint access parallel*) and not modifying shared memory during logically read-only operations (*read parallel*), and hence typically run in parallel on multi-processor machines. Finally, a low *operation footprint* (shared memory touched per operation) gives greater throughput under stress by easing pressure on the memory subsystem.

Our results reflect this, demonstrating performance comparable with the best existing designs in all tested cases. On highly-parallel workloads with many updates, our algorithm ran 35% faster; while a single-threaded run with mostly read-only operations was the worst case, running 40% slower than the best existing design.

Proof of correctness and progress properties can be found in Appendix A.

2 Background

2.1 Non-blocking Progress Guarantees

Data structures are easiest to implement when accessed in isolation, but general schemes for enforcing that isolation — for instance, using mutual exclusion locks — typically result in poor scalability and robustness in the face of contention and failure. Concurrent algorithms that avoid mutual exclusion are generally *non-blocking*: suspension of any subset of threads will not prevent forward progress by the rest of the system.

The weakest non-blocking guarantee is *obstruction-freedom*: if at any time a thread runs in isolation, it will complete its operation within a bounded number of steps. This precludes mutual exclusion, as suspension of a lock-holding thread will prevent others waiting on that lock from making progress. *Lock-freedom* combines this with guaranteed throughput: any active thread taking a bounded number of steps ensures global progress. Unfortunately, creating *practical* non-blocking forms of even simple data structures is notoriously difficult.

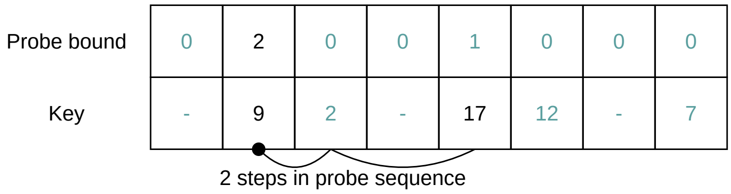
2.2 Related Work

Externally-chained hashtables store each bucket's collisions[1] in a list. Michael introduced the first practical lock-free hashtables based on external chaining with linked lists [8]. Shalev and Shavit described *split-ordered lists* that allow the number of buckets to vary dynamically [9]. Fraser detailed lock-free skip-lists and binary search trees [2]. Recently, Lea has contributed a high-performance, scalable, *lock-based*, externally-chained hashtable design to the latest version of Java (5.0), which avoids locking on most read-operations, preserving read-parallelism.

All of the above designs rely on an out-of-band garbage collector. Michael reported that reference counting was unacceptably slow for this purpose as it did not preserve read-parallelism; he proposed using safe memory reclamation [7] to get a strictly bounded memory overhead. Fraser used a simple low-overhead garbage collection scheme, *epoch-based reclamation*, where all threads maintain a current epoch, and memory is reclaimed only after all epochs change; this has a potentially unbounded memory footprint, and a large one in practice.

Tombstones are the traditional means of handling deletion in an open addressed hashtable [3], but cause degenerate search times in the face of a random workload with frequent deleting. Martin and Davis [5] proposed using periodic table replication to limit

[1] We refer to a key stored outside its primary bucket as a *collision*.



Probe bound	0	2	0	0	1	0	0	0
Key	-	9	2	-	17	12	-	7

Figure 1: Bounds on collision indices for a hash table holding keys $\{3, 7, 9, 12, 17\}$. Hash function is (key mod 8), probe sequence is quadratic $[\frac{1}{2}(i^2+i)]$. Key 17 is stored two steps along the probe sequence for bucket 1, so the probe bound is 2.

tombstone growth, relying on garbage collection to reclaim old tables. More recently, Gao *et al.* [1] presented a design with in-built garbage collection.

Both designs limit tombstone reuse to reinsertions of the old key, to achieve linearizability, and do not address the issue of storing multi-word keys directly in the table. The rest of our paper presents solutions to these problems, which we believe are compatible with the replication algorithms already proposed.

3 Memory-Management-Free Open Addressing

Each bucket in our hashtable stores a bound on its collisions' indices in the probe sequence (Figure 1). When running in isolation, a reader follows the probe sequence this number of steps before terminating; an insert that collides raises the bound if necessary; and an erase that empties the last bucket in this truncated probe sequence searches back for the previous collision and decreases the bound accordingly.

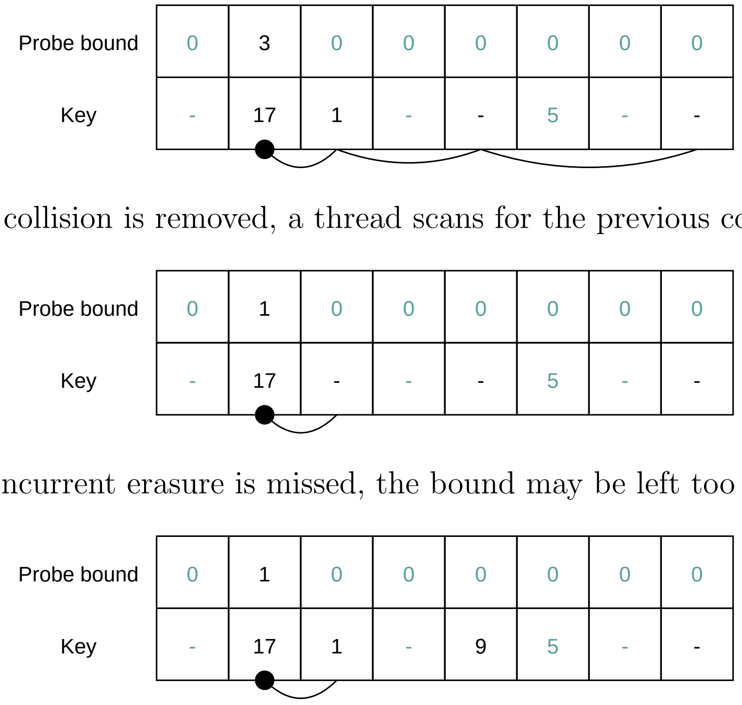
We make this safe for concurrent use in two steps, first maintaining each bucket's bound in Section 3.1, then ensuring keys are not duplicated in Section 3.2.

3.1 Bounding Searches

Maintaining the probe bounds concurrently is complicated by the need to lower them: simply scanning the probe sequence for the previous collision and swapping it into the bound field may result in the bound being too large if the collision is removed, slowing searches, or too small if another collision is inserted, violating correctness (Figure 2).

In order to keep the bounds correct during erasures, we use a *scanning phase* during which the thread erasing the last collision in the probe sequence searches through the previous buckets to compute the new bound (lines 18–22). A thread announces that it is in this phase by setting a *scanning bit* to true (line 18); this bit is held in the same word as the bound itself, so both fields are updated atomically.

Dealing with insertions is now easy: they atomically clear the scanning bit and raise the bound if necessary (lines 9–12). Deletions also clear the scanning bit (line 16), but are complicated by the scanning phase. We rely on the fact that at most one thread can be in the process of erasing a given collision, and that threads only start scanning when erasing the last collision in the probe sequence. The collision's index value thus identifies the scanning thread and, if it is still present as the bound when scanning completes, and if the scanning bit is still set, we know there have been no concurrent updates (line 22). Otherwise, we retry the scanning phase.



After a collision is removed, a thread scans for the previous collision.

If a concurrent erasure is missed, the bound may be left too large.

Worse, if a concurrent insertion is missed, the bound may be made too small.

Figure 2: Problems maintaining a shared bound after a collision is removed from the end of the probe sequence.

Given a lock-free atomic compare-and-swap (CAS) function, the pseudocode in Figure 3 is lock-free. We represent the packing of an int and a bit into a machine word with the $\langle ., . \rangle$ operator.

3.2 Inserting and Removing Keys

Inserting and removing keys concurrently is complicated by the lack of a pre-determined bucket for any given key. Inserting into the first empty bucket is not sufficient because, as Figure 4 shows, a concurrent erasure may alter which bucket is ‘first’, and a key may be duplicated. If duplicate keys are allowed in the table, concurrent key erasure becomes impractical.

To ensure uniqueness, we split insertions into three stages (Figure 5). First, a thread reserves an empty bucket and publishes its attempt by storing the key it is inserting, along with an ‘inserting’ flag. Next, the thread checks the other positions in the probe sequence for that key, looking for other threads with ‘inserting’ entries, or for a completed insertion of the same key. If it finds another insertion in progress in a bucket then it changes that bucket’s state to ‘busy’, stalling the other insertion at that point in time. If it finds another completed insertion of the same key, then its own insertion has failed: it empties its bucket and returns ‘false’. In the final stage, it attempts to finish its own insert, changing the ‘inserting’ flag in its bucket to ‘member’. It must do this with a CAS instruction so that it fails if stalled by another thread; if stalled, the thread republishes its attempt and restarts the second stage.

```
class Set {
  word bounds[size] // ⟨bound,scanning⟩

  void InitProbeBound(int h):
    bounds[h] := ⟨0,false⟩

  int GetProbeBound(int h): // Maximum offset of any collision in probe seq.
    ⟨bound,scanning⟩ := bounds[h]
    return bound

  void ConditionallyRaiseBound(int h, int index): // Ensure maximum ≥ index
    do
      ⟨old_bound,scanning⟩ := bounds[h]
      new_bound := max(old_bound,index)
    while ¬CAS(&bounds[h],⟨old_bound,scanning⟩,⟨new_bound,false⟩)

  void ConditionallyLowerBound(int h, int index): // Allow maximum < index
    ⟨bound,scanning⟩ := bounds[h]
    if scanning = true
      CAS(&bounds[h],⟨bound,true⟩,⟨bound,false⟩)
    if index > 0 // If maximum = index > 0, set maximum < index
      while CAS(&bounds[h],⟨index,false⟩,⟨index,true⟩)
        i := index-1 // Scanning phase: scan cells for new maximum
        while i > 0 ∧ ¬DoesBucketContainCollision(h, i)
          i--
        CAS(&bounds[h],⟨index,true⟩,⟨i,false⟩)
```

Figure 3: Per-bucket probe bounds (continued below)

The pseudocode in Figure 6 is obstruction-free. Each bucket contains a four-valued state, one of *empty*, *busy*, *inserting* or *member*, and, for the latter two states, a key. The key and state must be modified atomically; we use the $\langle ., . \rangle$ operator to represent packing them into a single word. A key k is considered inserted if some bucket in the table contains $\langle k, member \rangle$. The $Hash$ function selects a bucket for a given key. The three insertion stages can be found in lines 42–50, 51–60 and 61, respectively.

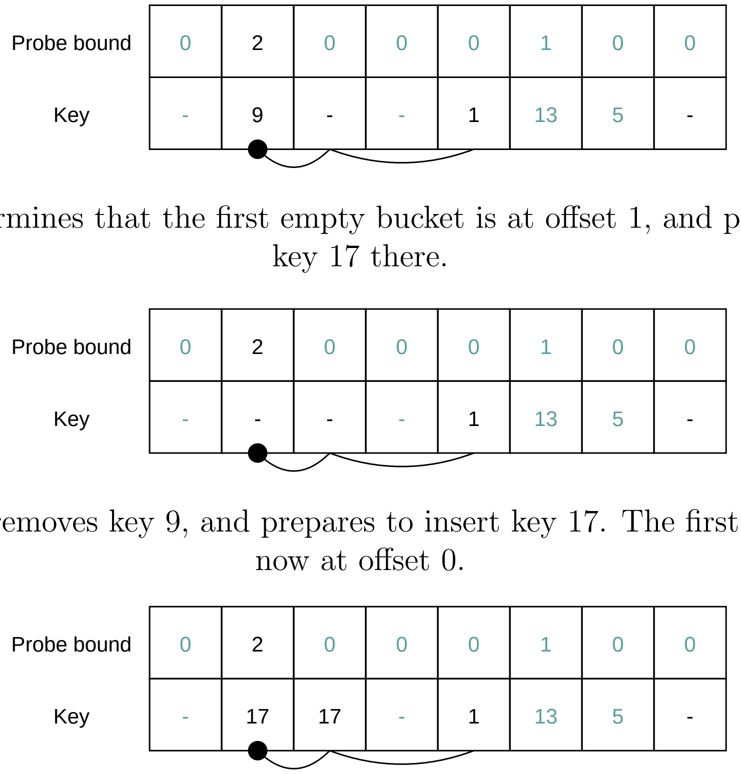
Unlike Martin and Davis' approach [5], deleted buckets are immediately free for arbitrary reuse, so table replication is not needed to clear out tombstones. The algorithm preserves read parallelism and, assuming disjoint keys hash to separate memory locations, disjoint access parallelism. In the expected case where the bucket contains no collisions, the operation footprint is two words — a single cache line if buckets and bounds are interleaved.

3.3 Extensions: Lock-Freedom and Multi-word Keys

We now turn to two flaws in the above algorithm. The first is that concurrent insertions may live-lock, each repeatedly stalling the other. One solution is to use an out-of-line contention manager: Scherer and Scott have described many suitable for use in any obstruction-free algorithm [10], which are easy to adopt. Another solution, which we cover in more detail as it is a non-trivial problem, is to make the algorithm lock-free.

The standard method of achieving lock-freedom is to allow operations to assist as well as obstruct each other. As given, however, the hash table cannot support concurrent assistance, as Figure 7 demonstrates: a cell's contents can change arbitrarily before returning to a previous state, allowing a CAS to succeed incorrectly. This is known as the ABA problem, and we return to it in a moment.

The second problem is storing keys larger than a machine word: in the algorithm as given, this requires a multi-word CAS, which is not generally available. However, we note


Probe bound 0 2 0 0 0 1 0 0
Key - 9 - - 1 13 5 -
Probe bound 0 2 0 0 0 1 0 0
Key - - - - 1 13 5 -
Probe bound 0 2 0 0 0 1 0 0
Key - 17 17 - 1 13 5 -

One thread determines that the first empty bucket is at offset 1, and prepares to insert key 17 there.

Another thread removes key 9, and prepares to insert key 17. The first empty bucket is now at offset 0.

The two threads now insert, creating a duplicate of the key.

Figure 4: Problems concurrently inserting keys

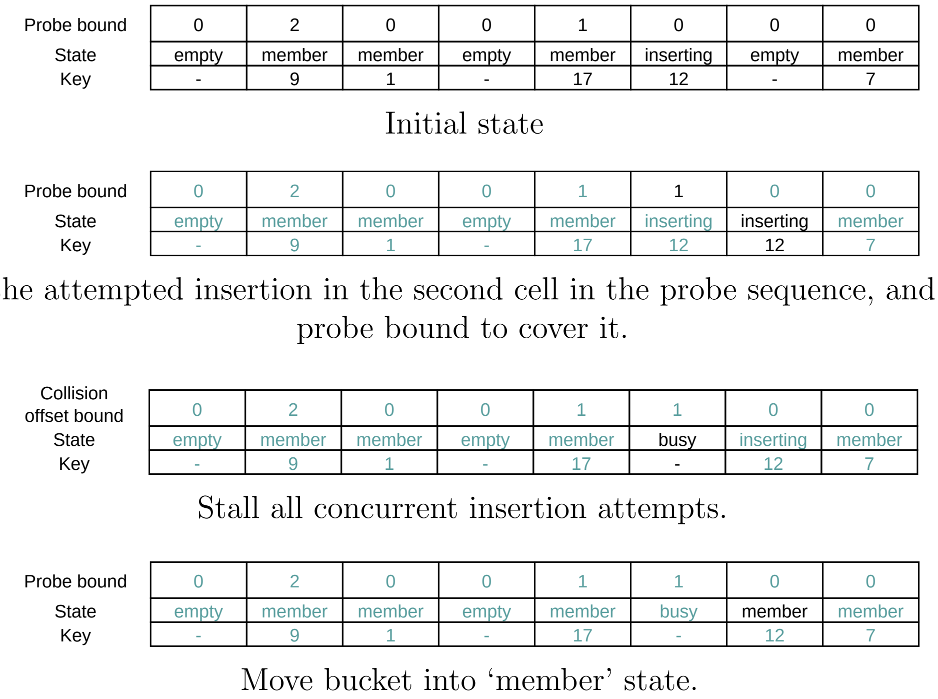
that a cell's key is only ever modified by a single writer, when the cell is in busy state. This means we only need to deal with concurrent single-writer multiple-reader access to the cell, rather than provide a general multi-word atomic update. We can therefore use Lamport's version counters [4] (Figure 8).

If a cell's state is stored in the same word as its version count, the ABA problem is circumvented, allowing threads to assist concurrent operations. This lets us create a lock-free insertion algorithm (diagram in Figure 9, pseudo-code in Figure 10).

Each bucket contains: a version count; a state field, one of *empty*, *busy*, *collided*, *visible*, *inserting* or *member*; and a key field, publically readable during the latter three stages. The version count and state are maintained so that no state (except busy) will recur with the same version.

As before, a thread finds an empty bucket and moves it into 'inserting' state (lines 65–76), and checks the probe sequence for other threads with 'inserting' entries, or a completed insertion of the same key (lines 86–106). However, if multiple 'inserting' entries are found, the earliest in the probe sequence is left unaltered, and the others moved into 'collided' state. When the whole probe sequence has been scanned and all contenders removed, the earliest entry is moved into 'member' state (line 105) and the insertion concludes (lines 78–83).

This version of the hashtable is lock-free.



Probe bound	0	2	0	0	1	0	0	0
State	empty	member	member	empty	member	inserting	empty	member
Key	-	9	1	-	17	12	-	7

Initial state

Probe bound	0	2	0	0	1	1	0	0
State	empty	member	member	empty	member	inserting	inserting	member
Key	-	9	1	-	17	12	12	7

Publish the attempted insertion in the second cell in the probe sequence, and raise the probe bound to cover it.

Collision offset bound	0	2	0	0	1	1	0	0
State	empty	member	member	empty	member	busy	inserting	member
Key	-	9	1	-	17	-	12	7

Stall all concurrent insertion attempts.

Probe bound	0	2	0	0	1	1	0	0
State	empty	member	member	empty	member	busy	member	member
Key	-	9	1	-	17	-	12	7

Move bucket into ‘member’ state.

Figure 5: Inserting key 12

3.4 Open Problems: Dynamic Growth and Key Replacement

If the set population approaches the number of buckets, we must replicate into a larger table. The Gao *et al.* [1] replication algorithm may be adaptable for this purpose. No aggregate time or memory cost is incurred on operations, as if the population stabilises, no further replications are required. Assuming each new table doubles in size, discarding the old table after growth is a memory overhead no greater than the final size of the table.

Even if a garbage collector is running, the bounded memory footprint provides several advantages. Many collectors are only activated when memory becomes scarce, so will benefit from less memory usage. Lacking pointers, no costly read or write barriers are needed to ensure memory is not leaked. Finally, the small number of memory allocations needed helps avoid any synchronization the allocator code may contain. The performance and latency benefits of these will depend on the memory management algorithms used.

As given, the algorithm cannot implement a dictionary, storing a value with each key, as there is no way to replace keys.

We hope to report these modifications in future work.

4 Results

In order to assess the performance of our new obstruction-free hashtable, we implemented a range of designs from the literature: Michael’s ‘dynamic lock-free hashtable’, which uses external chains to manage collisions and safe-memory-reclamation (MM-SMR) to manage storage, a variant of Michael’s design using epoch-based garbage collection (MM-Epoch), a further variant of Michael’s design using reference counting (MM-RC), and

```
  word buckets[size] // ⟨key,state⟩

  word* Bucket(int h, int index): // Size must be a power of 2
    return &buckets[(h + index*(index+1)/2) % size] // Quadratic probing

  bool DoesBucketContainCollision(int h, int index):
    ⟨k,state⟩ := *Bucket(h,index)
    return (k ≠ - ∧ Hash(k) = h)

public:
  void Init():
    for i := 0 .. size-1
      InitProbeBound(i)
      buckets[i] := ⟨-,empty⟩

  bool Lookup(Key k): // Determine whether k is a member of the set
    h := Hash(k)
    max := GetProbeBound(h)
    for i := 0 .. max
      if *Bucket(h,i) = ⟨k,member⟩
        return true
    return false

  bool Insert(Key k): // Insert k into the set if it is not a member
    h := Hash(k)
    i := 0 // Reserve a cell
    while ¬CAS(Bucket(h,i), ⟨-,empty⟩, ⟨-,busy⟩)
      i++
      if i ≥ size
        throw "Table full"
    do // Attempt to insert a unique copy of k
      *Bucket(h,i) := ⟨k,inserting⟩
      ConditionallyRaiseBound(h,i)
      max := GetProbeBound(h) // Scan through the probe sequence
      for j := 0 .. max
        if j ≠ i
          if *Bucket(h,j) = ⟨k, inserting⟩ // Stall concurrent inserts
            CAS(Bucket(h,j), ⟨k,inserting⟩, ⟨-,busy⟩)
          if *Bucket(h,j) = ⟨k,member⟩ // Abort if k already a member
            *Bucket(h,i) := ⟨-,busy⟩
            ConditionallyLowerBound(h,i)
            *Bucket(h,i) := ⟨-,empty⟩
            return false
    while ¬CAS(Bucket(h,i), ⟨k,inserting⟩, ⟨k,member⟩)
    return true

  bool Erase(Key k): // Remove k from the set if it is a member
    h := Hash(k)
    max := GetProbeBound(h) // Scan through the probe sequence
    for i := 0 .. max
      if *Bucket(h,i) = ⟨k,member⟩ // Remove a copy of ⟨k, member⟩
        if CAS(Bucket(h,i), ⟨k,member⟩, ⟨-,busy⟩)
          ConditionallyLowerBound(h,i)
          *Bucket(h,i) := ⟨-,empty⟩
          return true
    return false
}
```

Figure 6: Obstruction-free set (continued from Figure 3)

State	empty	inserting
Key	-	12

A single thread is about to complete its insertion of key 12. The next step is to atomically move the cell from inserting to member state.

State	empty	member
Key	-	12

The thread is suspended, and its insertion assisted to completion by another thread.

State	member	inserting
Key	12	12

The key is now removed, and two other threads are concurrently attempting to reinsert key 12. One has just succeeded, and the other is about to remove itself. If the first thread wakes up at this point, it will still atomically move the cell from inserting to member state, duplicating key 12.

Figure 7: Problems assisting concurrent operations

Shalev and Shavit's 'split-ordered lists' using epoch-based garbage collection (SS-Epoch). We also tested Lea's lock-based hashtable design, again using epoch-based collection. Since performance depends on the locking algorithm and the level of granularity (number of locks), we used a basic spinlock and the MCS lock [6] at different granularities. We compared these against our new design, as presented in Figures 3, 8 and 10 (PH).

Our benchmark is parameterized by the number of concurrent threads and by the range of key values used. We present results for 1–12 threads (running on a Sun Fire V880 with eight 900MHz UltraSPARC-III CPUs) and with 2^{15} keys chosen from $[0, 2^{15}M)$, M = 2 or 10. Each update step consists of removing a key then inserting another; finding keys and empty slots is done by trial-and-repetition, choosing candidates uniformly at random, giving $\frac{M^2}{M-1}$ searches on average for each update step. This was designed to avoid hashtable resizing, which simplifies our algorithm, as well as allowing a fine locking granularity and greater read-parallelism in Lea's, but which unfortunately negates the benefit of split-ordered lists.

Each trial lasted ten seconds, after a three second warm-up period to fill caches, and trials were repeated 20 times, interleaved to avoid short-lived anomalies, to obtain a 90% confidence interval. Our results are shown in Figure 11.

MM-Epoch and MM-SMR consistently outperform MM-RC and SS-Epoch (which, for clarity, are not shown in the results), thanks to low overhead and read-parallelism. Below 8 threads, DL-Epoch performs best with low-overhead spinlocking, avoiding the high cost of spinning with a fine locking granularity.

Searching for a key that is not in the table requires two memory accesses for the PH algorithm, but only one for all others tested. In the absence of contention, this is clearly visible in the results. Applications with a higher lookup hit rate would lower this cost. However, in all test with at least four threads, PH outperforms the other designs; this can largely be attributed to touching fewer cachelines (one rather than two) in the common-case code path for update operations — inter-processor cacheline exchange dominates

```
    struct BucketT {
        word vs // ⟨version,state⟩
        Key key
    } buckets[size]
    word buckets[size] // ⟨key,state⟩

    BucketT* Bucket(int h, int index): // Size must be a power of 2
        return &buckets[(h + index*(index+1)/2) % size] // Quadratic probing

    bool DoesBucketContainCollision(int h, int index):
        ⟨version1,state1⟩ := Bucket(h,index)→vs
        if state1 = visible ∨ state1 = inserting ∨ state1 = member
            if Hash(Bucket(h,index)→key) = h
                ⟨version2,state2⟩ := Bucket(h,index)→vs
                if state2 = visible ∨ state2 = inserting ∨ state2 = member
                    if version1 = version2
                        return true
        return false

public:
    void Init():
        for i := 0 .. size-1
            InitProbeBound(i)
            buckets[i].vs := ⟨0,empty⟩

    bool Lookup(Key k): // Determine whether k is a member of the set
        h := Hash(k)
        max := GetProbeBound(h)
        for i := 0 .. max
            ⟨version,state⟩ := Bucket(h,index)→vs // Read cell atomically
            if state = member ∧ Bucket(h,index)→key = k
                if Bucket(h,index)→vs = ⟨version,member⟩
                    return true
        return false

    bool Erase(Key k): // Remove k from the set if it is a member
        h := Hash(k)
        max := GetProbeBound(h)
        for i := 0 .. max
            ⟨version,state⟩ := Bucket(h,index)→vs // Atomically read/update cell
            if state = member ∧ Bucket(h,index)→key = k
                if CAS(Bucket(h,i)→vs, ⟨version,member⟩, ⟨version,busy⟩)
                    ConditionallyLowerBound(h,i)
                    Bucket(h,i)→vs := ⟨version+1,empty⟩
                    return true
        return false
```

Figure 8: Version-counted derivative of Figure 6 (continued in Figure 10)

Probe bound	0	2	0	0	1	0	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	inserting	empty	member
Key		9	1		17	12		7

Initial state

Probe bound	0	2	0	0	1	1	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	inserting	inserting	member
Key		9	1		17	12	12	7

Write key and raise probe sequence bound

Probe bound	0	2	0	0	1	1	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	inserting	collided	member
Key		9	1		17	12	12	7

Earlier ‘inserting’ entry found; move bucket into ‘collided’ state.

Probe bound	0	2	0	0	1	1	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	member	collided	member
Key		9	1		17	12	12	7

Assist completion of earlier entry

Probe bound	0	2	0	0	1	0	0	0
Version	18	2	3	6	4	3	25	7
State	empty	member	member	empty	member	member	empty	member
Key		9	1		17	12		7

Empty bucket, lower probe sequence bound and return `false`.

Figure 9: Inserting key 12 (lock-free algorithm)

```
  bool Insert(Key k): // Insert k into the set if it is not a member
    h := Hash(k)
    i := -1 // Reserve a cell
    do
      if ++i ≥ size
        throw "Table full"
      ⟨version,state⟩ := Bucket(h,i)→vs
    while ¬CAS(&Bucket(h,i)→vs, ⟨version,empty⟩, ⟨version,busy⟩)
    Bucket(h,i)→key := k
    while true // Attempt to insert a unique copy of k
      *Bucket(h,i)→vs := ⟨version,visible⟩
      ConditionallyRaiseBound(h,i)
      *Bucket(h,i)→vs := ⟨version,inserting⟩
      r := Assist(k,h,i,version)
      if Bucket(h,i)→vs ≠ ⟨version,collided⟩
        return true
      if ¬r
        ConditionallyLowerBound(h,i)
        Bucket(h,i)→vs := ⟨version+1,empty⟩
        return false
      version++

private:
  bool Assist(Key k,int h,int i,int ver_i): // Attempt to insert k at i
    // Return true if no other cell seen in member state
    max := GetProbeBound(h) // Scan through probe sequence
    for j := 0 .. max
      if i ≠ j
        ⟨ver_j,state_j⟩ := Bucket(h,j)→vs
        if state_j = inserting ∧ Bucket(h,j)→key = k
          if j < i // Assist any insert found earlier in the probe sequence
            if Bucket(h,j)→vs = ⟨ver_j,inserting⟩
              CAS(&Bucket(h,i)→vs, ⟨ver_i,inserting⟩, ⟨ver_i,collided⟩)
              return Assist(k,h,j,ver_j)
          else // Fail any insert found later in the probe sequence
            if Bucket(h,i)→vs = ⟨ver_i,inserting⟩
              CAS(&Bucket(h,j)→vs, ⟨ver_j,inserting⟩, ⟨ver_j,collided⟩)
        ⟨ver_j,state_j⟩ := Bucket(h,j)→vs // Abort if k already a member
        if state_j = member ∧ Bucket(h,j)→key = k
          if Bucket(h,j)→vs = ⟨ver_j,member⟩
            CAS(&Bucket(h,i)→vs,⟨ver_i,inserting⟩,⟨ver_i,collided⟩)
            return false
    CAS(&Bucket(h,i), ⟨ver_i,inserting⟩, ⟨ver_i,member⟩)
    return true
}
```

Figure 10: Lock-free insertion algorithm (continued from Figure 8)

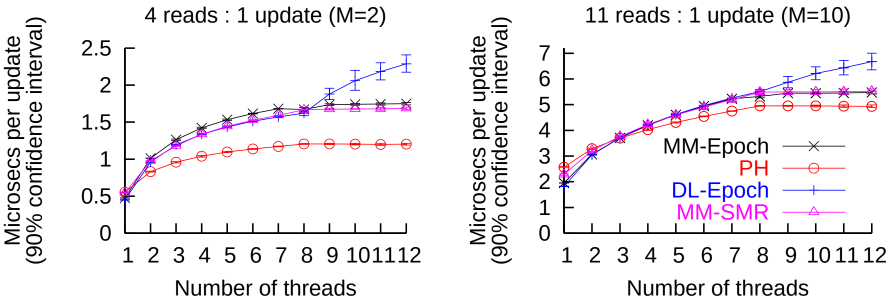


Figure 11: Performance on 8-way SPARC machine

runtime in massively parallel workloads. Applications with much larger, multi-cacheline keys would lose most of this advantage, and may favour an externally-chained scheme to lower the memory footprint of empty buckets.

5 Conclusions

We have presented a lock-free, disjoint-access and read parallel set algorithm based on open addressing, with no need for garbage collection, and touched upon removing population constraints. It has high straight-line speeds and a low operation footprint leading to excellent performance, matching and besting state-of-the-art external-chaining implementations in the tests we performed.

We wish to thank Sun Microsystems, Inc. for donating the SPARC v880 server on which this work was evaluated, and the University of Rochester, New York, for hosting it.

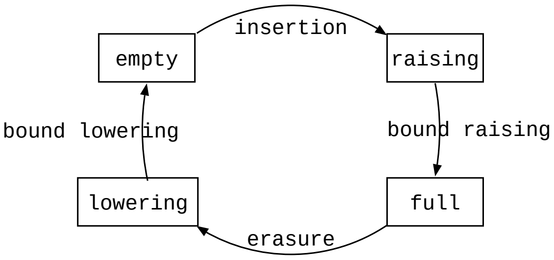


Figure 12: Shared bound state-machine

A Proofs

In the following, we use t_c and t_r for the call and return times of a function, respectively, and t_n for the linearization point of the last pass through line n during that call.

A.1 Shared Bound: Proof of Correctness

For simplicity, we consider a single bucket h, with a probe sequence bound (bound) and a scanning bit (scanning). We model the probe sequence at time t as an infinite sequence $(b_i^t)_{i \geq 0}$ of state machines, as shown in Figure 12, where $b_i^t \in \{\text{empty}, \text{raising}, \text{full}, \text{lowering}\}$ $\forall i, t$, and where $b_i^0 = \text{empty}\ \forall i$.

The pre-condition for the *bound raising* transition for bucket i is that bound $\geq i$ and scanning = false, and for *bound lowering* that bound $\neq i$ and scanning = false; the *insertion* and *erasure* transitions are handled by the calling code.

Our correctness criteria consist of the following claims about the code, given certain restrictions on the calling code:

Criterion. ConditionallyRaiseBound(h,i) post-condition: $b_i^{t_r} = \text{full}$, assuming $b_i^t \in \{\text{raising}, \text{full}\}\ \forall t \in [t_c, t_r]$

Criterion. ConditionallyLowerBound(h,i) post-condition: $b_i^{t_r} = \text{empty}$, assuming $b_i^t \in \{\text{lowering}, \text{empty}\}\ \forall t \in [t_c, t_r]$

Criterion. GetProbeBound(h) returns $\max\Big(\Big\{i : b_i^{t_6} \in \{\text{full}, \text{lowering}\}\Big\} \cup \{0\}\Big)$

The following two conditions are assumed to hold:

Collision condition. DoesBucketContainCollision(h,i) returns `true` (resp. `false`) only if $b_i^t \in \{\text{raising}, \text{full}\}$ (resp. $\{\text{lowering}, \text{empty}\}$) holds for some $t \in [t_c, t_r]$, where DoesBucketContainCollision(h,i) is called at t_c and returns at t_r

Exclusivity condition. $|R_i^t \cup L_i^t| = 1\ \forall i, t$
where $R_i^t = \{\text{threads in ConditionallyRaiseBound(h,i) at time t}\}$
and $L_i^t = \{\text{threads in ConditionallyLowerBound(h,i) at time t}\}$

First Scanning Lemma. The scanning bit is cleared whenever the bound changes, and the scanning bit is only set at line 18.

Proof. Examination of the code. Only lines 12, 16, 18 and 22 change the bound and scanning bit, and none violate the lemma.

Post-condition on ConditionallyRaiseBound(h,i). $b_i^{t_r} = \text{full}$, assuming $\forall t \in [t_c, t_r]$, $b_i^t \in \{\text{raising}, \text{full}\}$

Proof. The CAS at line 12 must succeed before t_r, and a successful CAS ensures $\text{bound}^{t_{12}} \geq i$ and $\text{scanning}^{t_{12}} = \text{false}$, triggering the *bound raising* state transition if bucket i is in raising state. By assumption, there are no erasure transitions, so the post-condition must hold.

Lemma 2. $\exists t \in [t_{14}, t_{16}]$ s.t. $\text{scanning}^t = \text{false}$

Proof. If $\text{scanning}^{t_{14}} = \text{true}$ and $\text{scanning}^{t_{16}-} = \text{true}$, either $\text{bound}^{t_{14}} = \text{bound}^{t_{16}-}$ and the CAS at line 16 succeeds, ensuring $\text{scanning}^{t_{16}} = \text{false}$, or we appeal to the First Scanning Lemma. The lemma follows.

Post-condition on Line 22. $\langle \text{bound}, \text{scanning} \rangle^{t_{22}} \neq \langle \text{index}, \text{true} \rangle$

Second Scanning Lemma. If bound $= \langle \text{index}, \text{true} \rangle$, some thread is executing ConditionallyLowerBound(h,index) lines 19–22.

Proof. Only line 18 can set bound $= \langle \text{index}, \text{true} \rangle$; if this occurs, lines 19–22 will be executed. The post-condition on line 22 then implies the lemma.

Pre-condition on Line 18. $\langle \text{bound}, \text{scanning} \rangle^{t_{22}} \neq \langle \text{index}, \text{true} \rangle$

Proof. Consequence of the exclusivity condition and the Second Scanning Lemma.

Lemma 6. If index > 0, $\exists t \in [t_{14}, t_r]$ s.t. $\text{bound}^t \neq \text{index}$ and $\text{scanning}^t = \text{false}$

Proof. By lemma 2, $\exists t_* \in [t_{14}, t_{16}]$ s.t. $\text{scanning}^{t_*} = \text{false}$. By the pre-condition on line 18, the loop of lines 18–22 will only end when $\text{bound}^{t_{18}} \neq \text{index}$. Thus, if $\text{bound}^{t_*} = \text{index}$, we can appeal to the First Scanning Lemma to find t; otherwise, $t = t_*$ satisfies the lemma.

Claim. ConditionallyLowerBound(h,i) post-condition: $b_i^{t_r} = \text{empty}$ assuming $b_i^t \in \{\text{lowering}, \text{empty}\}\ \forall t \in [t_c, t_r]$

Proof. By lemma 6, the *bound lowering* state transition must occur during ConditionallyLowerBound(h,i) if bucket i is in lowering state. The pre- and during-conditions prevent an insertion transition, so the post-condition must hold.

Theorem. $\forall i, t,\ \left(b_i^t \in \{\text{empty}, \text{raising}\} \Rightarrow \text{bound}^t \neq i\right)$, $\left(b_i^t \in \{\text{full}, \text{lowering}\} \Rightarrow \text{bound}^t \geq i\right)$ and $\text{bound}^t \geq 0$

Corollary. $\text{bound}^t = \mathit{max}\left(\{i : b_i^t \in \{\text{full}, \text{lowering}\}\} \cup \{0\}\right)\ \forall t$

Corollary. GetProbeBound(h) returns $\max\left(\left\{i : b_i^{t_6} \in \{\text{full}, \text{lowering}\}\right\} \cup \{0\}\right)$

Lemma 7. If bucket i has a *bound raising* transition at time t, $\text{bound}^t \geq i$.

Lemma 8. If bucket i has a *bound lowering* transition at time t, $\text{bound}^t \neq i$.

Proofs. Both lemmas follow immediately from the pre-conditions of the state transitions.

Lemma 9. Bucket j cannot remain in raising state at time t if $\text{bound}^{t-} \neq \text{bound}^t$ and $\text{bound}^t > j$.

Lemma 10. Bucket j cannot remain in lowering state at time t if $\text{bound}^{t-} \neq \text{bound}^t$ and $\text{bound}^t \neq j$.

Proofs. Both lemmas follow immediately from the pre-conditions of the state transitions and the First Scanning Lemma.

Post-condition on Line 12. $\text{bound}^{t_{12}} >= \text{bound}^{t_{12}-}$

Lemma 12. If a call to ConditionallyRaiseBound(h,i) alters bound at line 12 and the conditions of the Theorem hold at time $t_{12}-$, they hold at time t_{12}.

Proof. By Lemmas 7 and 8, we need only consider buckets that do not undergo a *bound raising* or *bound lowering* transition at time t_{12}. By Lemmas 9 and 10, this leaves all buckets in empty or full state, and any bucket $j > \text{bound}^{t_{12}}$ in raising state.

$b_j^{t_{12}-} \in \{\text{raising}, \text{full}\}$ by the precondition on ConditionallyRaiseBound(h,i), so any bucket j in empty state satisfies $j \neq \text{bound}^{t_{12}}$. If $b_j^{t_{12}-} = \text{full}$, then by hypothesis $\text{bound}^{t_{12}-} > j$, so by the post-condition on line 12, $\text{bound}^{t_{12}} > j$. Finally, any j where $b_j^{t_{12}-} = b_j^{t_{12}} = \text{raising}$ satisfies $j > \text{bound}^{t_{12}}$ as already stated.

Hence the conditions of the Theorem hold at time t_{12}.

Third Scanning Lemma. If $\langle \text{bound}, \text{scanning} \rangle^{t_{22}-} = \langle \text{index}, \text{true} \rangle$, no buckets have undergone either *bound raising* or *bound lowering* transitions on the interval $[t_{19}, t_{22})$.

Proof. Consequence of exclusivity condition, Second Scanning Lemma and preconditions of the state transitions.

Post-conditions on Line 22. If the CAS succeeds, $\text{bound}^{t_{22}} < \text{bound}^{t_{22}-}$, $\forall j \in \left(\text{bound}^{t_{22}}, \text{bound}^{t_{22}-}\right]$, $\exists t \in [t_{19}, t_{22}]$ s.t. $b_j^t \in \{\text{lowering}, \text{empty}\}$ and $\text{bound}^{t_{22}} = \mathtt{i} > 0 \Rightarrow b_{\mathtt{i}}^{t_{20}} \in \{\text{raising}, \text{full}\}$.

Lemma 15. If a call to ConditionallyLowerBound(h,i) alters bound at line 22 at time t_{22} and the conditions of the Theorem hold at time $t_{22}-$, they hold at time t_{22}.

Proof. By Lemmas 7 and 8, we need only consider buckets that do not undergo a *bound raising* or *bound lowering* transition at time t_{22}. By Lemmas 9 and 10, this leaves all buckets in empty or full state, and any bucket $j > \text{bound}^{t_{22}}$ in raising state.

By the Third Scanning Lemma, any bucket in empty or full state at time t_{22} must have been so at time t_{19}, so by the post-conditions on line 22 and by hypothesis, $\forall i$, $b_i^{t_{22}} = \mathit{empty} \Rightarrow \text{bound}^{t_{22}} \neq i$ and $b_i^{t_{22}} = \mathit{full} \Rightarrow \text{bound}^{t_{22}} \geq i$. Finally, any j where $b_j^{t_{22}-} = b_j^{t_{22}} = \mathit{raising}$ satisfies $j > \text{bound}^{t_{22}}$ as already stated.

Hence the conditions of the Theorem hold at time t_{22}.

Proof of Theorem. By construction, the Theorem holds at time 0. We proceed by induction on the steps taken by each thread under some global ordering.

Suppose a thread executes an operation at time t', and the Theorem holds $\forall t < t'$. The theorem can only be false at time t' if a bucket i undergoes a *bound raising* or *bound lowering* transition, or if $\text{bound}^{t'-} \neq \text{bound}^{t'}$. By Lemmas 7 and 8, neither state transition will invalidate the theorem. The bound field is only altered by lines 12 and 22, and by Lemmas 12 and 15, neither line will invalidate the theorem.

Hence the Theorem holds at time t', and by induction for all t.

A.2 Shared Bound: Proof of Progress

GetProbeBound is trivially lock-free. We show that the remaining two functions in Figure 3 are lock-free, assuming correct behaviour by the calling code, using an amortization argument.

Inspection of the pseudocode reveals that failed CASes either result in local progress (lines 16 and 18) or result from a concurrent, successful CAS during a loop (lines 9–12 and 18–22); we therefore ignore failed CASes.

The scanning bit is assigned a credit of one when clear, and zero when set; lines 12, 16 and 22, when successful, may clear the scanning bit, and hence have an amortized cost of 2. Line 18 has no amortized cost when successful, as it always sets the scanning bit, and hence can be charged against the scanning bit's credit.

Claim. ConditionallyRaiseBound(h,i) and ConditionallyLowerBound(h,i) have amortized costs of at most 2 or 4 successful CAS operations, respectively, provided the code calling ConditionallyLowerBound(h,i) ensures $b_i^t \in \{\text{lowering}, \text{empty}\}$ $\forall t \in [t_c, t_r]$.

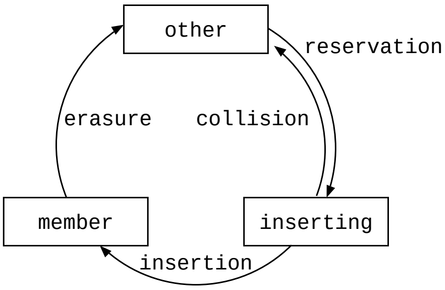


Figure 13: Uniqueness state-machine

Lemma. The CAS at line 22 will succeed at most once per call.

Proof. Let t^* be the time of the first successful CAS at line 22; this CAS must trigger the *bound lowering* state transition. Hence, by the Theorem above, $bound_t \neq i \ \forall t \in [t^*, t_r]$, and by inspection the loop of lines 18–22 will subsequently terminate without repeating.

Proof of Claim. ConditionallyRaiseBound(h,i) executes line 12 successfully exactly once, and hence has an amortized cost of 2. By inspection and the Lemma, ConditionallyLowerBound(h,i) will execute lines 16 and 22 successfully at most once, and hence has an amortized cost of no more than 4.

A.3 Sets: Proof of Uniqueness

For simplicity, we consider a single key k. We model the probe sequence for bucket $h = \texttt{Hash}(k)$ at time t as an infinite sequence $(s_i^t)_{i\geq 0}$ of state machines, as shown in Figure 13, where $s_i^t \in \{\text{inserting}, \text{member}, \text{other}\} \ \forall i, t$ and $s_i^0 = \text{other} \ \forall i$.

In both obstruction-free and lock-free sets, $b_j =$ either inserting or member if $\text{bound}_h \geq j$, bucket j holds key k and is in `inserting` or `member` state respectively, and $b_j = \text{other}$ in all other cases.

Uniqueness Criterion. $|\{i : b_i^t = \text{member}\}| \leq 1 \ \forall t$

Our correctness criterion relies on a single condition:

Collision Condition. If $\exists i, u, v \ (u < v)$ such that $s_i^{u-} = \text{other}$, $s_i^t = \text{inserting} \ \forall t \in [u, v)$ and $s_i^v = \text{member}$, then $\forall j \neq i$, $\exists t \in [u, v)$ with $s_j^t = \text{other}$.

Proof of Criterion. Suppose $\exists i, j, y$ with $i \neq j$ and $s_i^y = s_j^y = \text{member}$. The state-machine and starting conditions imply that $\exists u, v, w, x$ with $s_i^{u-} = \text{other}$, $s_i^t = \text{inserting} \ \forall t \in [u, v)$, $s_i^t = \text{member} \ \forall t \in [v, y]$, $s_j^{w-} = \text{other}$, $s_j^t = \text{inserting} \ \forall t \in [w, x)$ and $s_j^t = \text{member} \ \forall t \in [x, y]$. Without loss of generality, suppose $u \geq w$. Then by the collision condition, $\exists t \in [u, v)$ with $s_j^t = \text{other}$, but $t \geq u \geq w$ and we derive a contradiction.

Hence the uniqueness criterion holds.

In fact, this proof can be extended to prove a stronger claim:

Uniqueness Lemma. $\forall i, j, u, v (i \neq j, u \leq v)$, $s_i^u = \text{member} = s_j^v \Rightarrow \exists t \in (u, v)$ with $s_i^t \neq \text{member}$ and $s_j^t \neq \text{member}$.

Lookup Lemma. $\forall u, v$, $(\forall i, \exists t \in (u, v)$ s.t. $s_i^t \neq \text{member}) \Rightarrow \exists t \in (u, v)$ s.t. $n_t = 0$.

Proof. By the uniqueness lemma, $n_t = 1 \ \forall t \in (u, v) \Rightarrow \exists i$ s.t. $s_i^t = \text{member} \ \forall t \in (u, v)$. The result follows.

A.4 Obstruction-Free Set: Proof of Correctness

We wish to show the pseudo-code in Figures 3 and 6 maintain a logical set, S, of keys. For simplicity, we consider a single key k.

Definitions.
$n_t = |\{i : s_i^t = \text{member}\}|$
$k \in S_t \iff n_t = 1$
$I' = \{t : k \notin S_{t-}, k \in S_t\}$
$E' = \{t : k \in S_{t-}, k \notin S_t\}$
$L_s = \{\text{Calls to } \mathsf{Lookup(k)} \text{ that return true}\}$
$L_f = \{\text{Calls to } \mathsf{Lookup(k)} \text{ that return false}\}$
$I_s = \{\text{Calls to } \mathsf{Insert(k)} \text{ that return true}\}$
$I_f = \{\text{Calls to } \mathsf{Insert(k)} \text{ that return false}\}$
$E_s = \{\text{Calls to } \mathsf{Erase(k)} \text{ that return true}\}$
$E_f = \{\text{Calls to } \mathsf{Erase(k)} \text{ that return false}\}$
If x is a function call, t_c^x is the time it was called, t_r^x the time it returns, and t_n^x the last time it executed line n.

The code is correct iff it satisfies the following criteria:

Lookup Criterion. $\forall x \in L_s, \exists t \in [t_c^x, t_r^x]$ s.t. $k \in S_t$. $\forall x \in L_f, \exists t \in [t_c^x, t_r^x]$ s.t. $k \notin S_t$.

Insertion Criterion. $\exists f : I_s \to I'$, f a bijection, with $f(x) \in [t_c^x, t_r^x], k \notin S_{f(x)-}$ and $k \in S_{f(x)}$ $\forall x \in I_s$. $\forall x \in I_f, \exists t \in [t_c^x, t_r^x]$ s.t. $k \in S_t$.

Erasure Criterion. $\exists g : E_s \to E'$, g a bijection, with $g(x) \in [t_c^x, t_r^x], k \in S_{g(x)-}$ and $k \notin S_{g(x)}$ $\forall x \in E_s$. $\forall x \in E_f, \exists t \in [t_c^x, t_r^x]$ s.t. $k \notin S_t$.

Bound Lemma. $\text{bound}^t = i \Rightarrow s_j^t = \text{other}\ \forall j > i$

Proof. Immediate consequence of definitions.

Claim. The lookup criterion holds.

Proof. $\mathsf{Lookup(k)}$ returns true only if $\exists t, i$ $(t \in [t_c, t_r])$ s.t. $s_i^t = \text{member}$, which by the uniqueness criterion implies $n_t = 1$. Appealing to the bound lemma, $\mathsf{Lookup(k)}$ returns false only if $\forall i, \exists t \in [t_c, t_r]$ s.t. $s_i^t \neq \text{member}$, which by the lookup lemma implies $\exists t \in [t_c, t_r]$ s.t. $n_t = 0$.

Insert lemma. inserting → member state transitions occur only at line 61, at time t_{61}

Pre-condition on Line 57. $s_j^{t_{56}} = \text{member}$.

Pre-conditions on Line 62. $s_i^{t_{61}-} \neq \text{member}$, $s_i^{t_{61}} = \text{member}$ and $\forall j \neq i, \exists t \in [t_{49}, t_{61})$ s.t. $s_i^t \neq \text{member}$.

Proofs. Inspection of the pseudo-code, and appeal to the bound lemma.

Claim. The insertion criterion holds.

Proof. $\mathsf{Insert(k)}$ returns after passing through either lines 57–60, returning false, or line 62, returning true. The pre-condition on line 57 implies $n_{t_{56}} = 1$. The pre-conditions on line 62 satisfy the collision condition, and hence by the uniqueness lemma, $n_{t_{61}-} = 0$ and $n_{t_{61}} = 1$. Further, only one thread can succeed in the CAS at line 61 at time t_{61} for bucket i, and no other threads can succeed at that time for any other bucket by the uniqueness lemma; hence if f maps $x \in I_s$ to t_{61}^x, f is an injection. By the insert lemma, f is also a surjection, and hence a bijection from I_s to I' with the desired properties.

Erase lemma. member → other state transitions occur only at line 68, at time t_{68}.

Pre-conditions on Line 69. $s_i^{t_{68}-} = \text{member}$ and $s_i^{t_{68}} \neq \text{member}$.

Pre-condition on Line 72. $\forall i, \exists t \in [t_{65}, t_{72})$ s.t. $s_i^t \neq \text{member}$.

Claim. The erasure criterion holds.

Proof. Erase(k) returns after passing through either lines 69–71, returning true, or line 72, returning false. By the lookup lemma, the pre-condition on line 62 implies $\exists t \in [t_{65}, t_{72})$ s.t. $n_t = 0$. The pre-conditions on line 69 satisfy the requirements of the erasure condition. Further, only one thread can succeed in the CAS at line 68 at time t_{68} for bucket i, and no other threads can succeed for any other bucket at that time by the uniqueness lemma; hence if g maps a successful Erase(k) to its t_{68}, g is an injection. By the erase lemma, g is also a surjection, and hence a bijection from J to J' as desired.

A.5 Obstruction-Free Set: Proof of Progress

Both Lookup and Erase are lock-free, and hence obstruction-free, as the bounds returned by GetProbeBound cannot exceed the largest index of any bucket in the table. Insert(k) only repeats lines 48–61 if the ultimate CAS fails, which only occurs if the value written at line 49 is concurrently altered. In isolation the loop will terminate, making the function obstruction-free.

A.6 Lock-Free Set: Proof of Correctness

The proof of correctness for the lock-free set is identical in structure to that of the obstruction-free set, and will not be duplicated.

A.7 Lock-Free Set: Proof of Progress

Definitions.

$$\text{Let } e_t = \begin{cases} -1 & \text{if } \exists j \text{ s.t. } s_j^t = \text{member} \\ \infty & \text{if } \forall j, s_j^t = \text{other} \\ min\left\{j : s_j^t = \text{inserting}\right\} & \text{otherwise} \end{cases}$$

Let (t_i) be the increasing sequence s.t. $\{t_i\} = \{t : e_t \neq e_{t-}\}$; let $e_i = e_{t_i}$ for brevity.
If $e_i \in [0, \infty)$, let T_i be the last thread that put cell e_i into inserting state by time t_i.
Thread T running Insert(k) is said to *abort after time* t if $t \in [t_{76}, t_r]$ and T returns failure.

Precedence Lemma. $\exists i, u, v\ (u < v)$ s.t. $s_i^u = \text{other}, s_i^t = \text{inserting}\ \forall t \in [u, v), s_i^v = \text{other} \Rightarrow \exists j, t \in [u, v)$ s.t. either $s_j^t = \text{member}$ or $s_j^t = \text{inserting}$ and $j < i$.

Assist Lemma. If Assist(k,...) returns true, $\forall i\ \exists t \in [t_c, t_r]$ s.t. $s_i^t = \text{other}$. If Assist(k,...) returns false after executing line 76 at t_{76}, either $e_t = -1\ \forall t \in [t_{76}, t_r]$, or $\exists t \in [t_{76}, t_r]$ with $e_{t-} \neq e_t$.

Proof. Inspection of pseudo-code.

Sawtooth Lemma. $0 < e_{i-1} < e_i \Rightarrow e_{i-2} < e_{i-1}$.

Proof. $0 < e_{i-1} < e_i$ only if $s_{e_i}^{t_i} = \text{other}$ and $s_{e_i}^{t_i-} = \text{inserting}$. By the precedence lemma, $\exists t' < t_i$ s.t. $s_{e_i}^t = \text{inserting}\ \forall t \in [t', t_i)$, and $\exists j$ s.t. either $s_j^{t'} = \text{member}$ or $j < i$ and $s_j^{t'} = \text{inserting}$. Since $s_{e_i}^t = \text{inserting} \Rightarrow e_t \leq e_i$ by definition, $s_j^{t'} = \text{member} \Rightarrow e_j = -1$, and $s_j^{t'} = \text{inserting} \Rightarrow e_j < j$, it follows that $e_{t'} < j$. The lemma follows.

Increase Lemma. If $s_i < s_{i+1} < ... < s_j$, some i, j, then $|\{k \in [i, j) : T_k \text{ aborts after } t_k\}| \geq j - i - N$, where N is the number of threads.

Proof. $T_i = T_j$ only if thread T_i completed its first operation and started a new one. If $e_i < e_{i+1}$, thread T_i can only return from Insert(k) after retrying Assist(k,...) or failing. $|\{T_k : k \in [i, j)\}| \leq N$ gives the result.

Decrease Lemma. If $s_i > s_{i+1} > ... > s_j$, some i, j, then $|\{k \in [i, j) : T_k \text{ aborts after } t_k\}| \geq j - i - N$, where N is the number of threads.

Proof. $T_i = T_j$ only if thread T_i completed its first operation and started a new one. If $e_k < -1\ \forall k \in [i, j]$, thread T_i can only return from Insert(k) after retrying Assist(k,...) or failing. $|\{T_k : k \in [i, j)\}| \leq N$ gives the result.

Cost Lemma. $\left|\left\{(t, j) : t \in [t_i, t_{i+1}), s_j^{t-} = \text{inserting}, s_j^t = \text{other}\right\}\right| \leq 2N\ \ \forall i$ where $e_i \neq -1$

Proof. Consequence of assist lemma: observe that if any thread call Assist(k,...), either $e_t = -1$ for the entire duration, or the value of (e_t) must change, and every state transition other → inserting must be followed by a call to Assist(k,...).

Claim. The pseudo-code in Figure 10 is lock-free.

Proof. Let p_t be the *progress* at time t. Moving a cell from inserting state increments the progress counter, while each call to Insert(k) decrements it by $4N^2$. We wish to show that $p_t \leq 0\ \forall t$.

By the cost lemma, $p_{t_{i+1}} - p_{t_i} \leq 2N\ \forall i$. By the sawtooth, increase and decrease lemmas, e_t can change at most $2N$ times before an operation completes. The claim follows.

References

[1] Gao, H., Groote, J. and Hesselink, W. Almost Wait-Free Resizable Hashtables In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, April 2004, p.50a.

[2] Fraser, K. Practical Lock-Freedom. *University of Cambridge Computer Laboratory, Technical Report number 579*, February 2004.

[3] Knuth, D. The Art of Computer Programming. Part 3, Sorting and Searching. Addison-Wesley, 1973.

[4] Lamport, L. Concurrent Reading and Writing. In *Communications of the ACM*, 1977, pp.806-811.

[5] Martin, D. and Davis, R. A Scalable Non-Blocking Concurrent Hash Table Implementation with Incremental Rehashing. Unpublished manuscript, 1997.

[6] Mellor-Crummey, J. and Scott, M. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. In *ACM Transactions on Computer Systems*, Volume 9, Issue 1, February 1991, pp. 21–65.

[7] Michael, M. Safe Memory Reclamation for Dynamic Lock-Free Objects using Atomic Reads and Writes. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, July 2002, pp.21-30.

[8] Michael, M. High performance dynamic lock-free hash tables and list-based sets In *Proceedings of the 14th Annual Symposium on Parallel Algorithms and Architectures*, August 2002, pp.73-82.

[9] Shalev, O. and Shavit, N. Split-Ordered Lists: Lock-free Extensible Hash Tables. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, July 2003, pp.102-111.

[10] Scherer, W. and Scott, M. Contention Management in Dynamic Software Transactional Memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004, pp.70–79.

[11] Purcell, C. and Harris, T. Non-blocking Hashtables with Open Addressing. *University of Cambridge Computer Laboratory, Technical Report number 639*, September 2005.