Non-blocking Hashtables with Open Addressing

Chris Purcell[1] and Tim Harris[2]

[1] Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK.
Chris.Purcell@cl.cam.ac.uk
[2] Microsoft Research Ltd., Roger Needham Building, 7 JJ Thomson Avenue, Cambridge CB3 0FB, UK.
tharris@microsoft.com

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 multiprocessor 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 [11].

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[3] 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.

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

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 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.

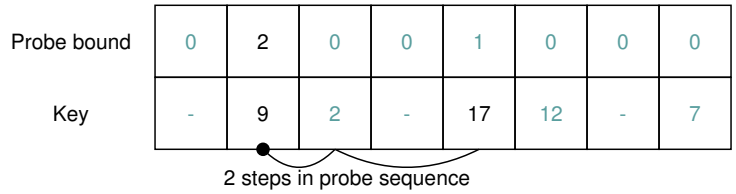
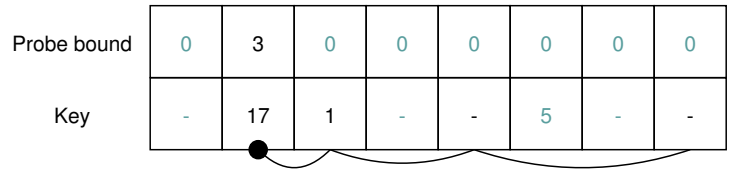
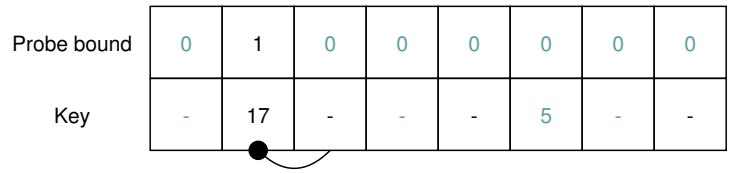


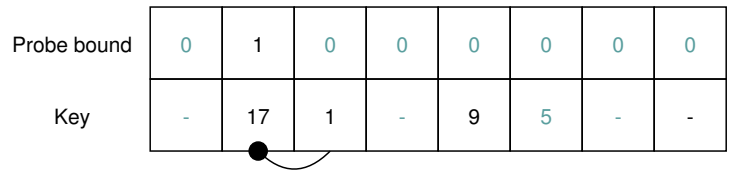
Fig. 1. Bounds on collision indices for a hash table holding keys $\{3, 7, 9, 12, 17\}$. Hash function is (key mod 8), probe sequence is quadratic $[\frac{1}{2}(i^2 + i)]$. Key 17 is stored two steps along the probe sequence for bucket 1, so the probe bound is 2.



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.

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

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.

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.

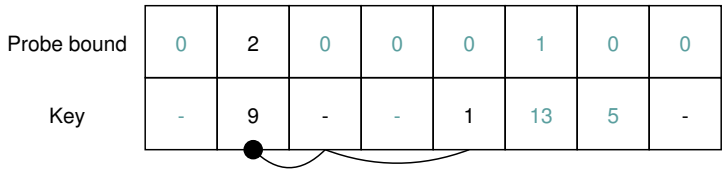
```
1   class Set {
      word bounds[size] // ⟨bound,scanning⟩
3     void InitProbeBound(int h):
        bounds[h] := ⟨0,false⟩
5     int GetProbeBound(int h): // Maximum offset of any collision in probe seq.
        ⟨bound,scanning⟩ := bounds[h]
7       return bound
      void ConditionallyRaiseBound(int h, int index): // Ensure maximum ≥ index
9       do
          ⟨old_bound,scanning⟩ := bounds[h]
11        new_bound := max(old_bound,index)
        while ¬CAS(&bounds[h],⟨old_bound,scanning⟩,⟨new_bound,false⟩)
13    void ConditionallyLowerBound(int h, int index): // Allow maximum < index
        ⟨bound,scanning⟩ := bounds[h]
15      if scanning = true
          CAS(&bounds[h],⟨bound,true⟩,⟨bound,false⟩)
17      if index > 0 // If maximum = index > 0, set maximum < index
          while CAS(&bounds[h],⟨index,false⟩,⟨index,true⟩)
19          i := index-1 // Scanning phase: scan cells for new maximum
            while i > 0 ∧ ¬DoesBucketContainCollision(h, i)
21            i--
            CAS(&bounds[h],⟨index,true⟩,⟨i,false⟩)
```

Fig. 3. Per-bucket probe bounds (continued below)

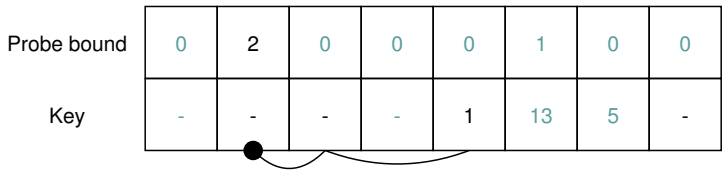
3.2 Inserting and Removing Keys

Inserting and removing keys concurrently is complicated by the lack of a predetermined 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.

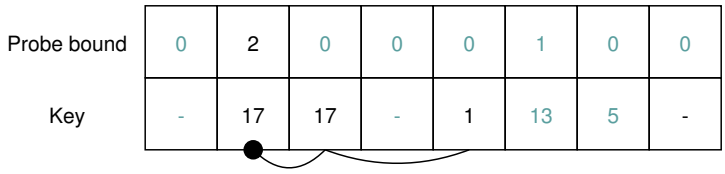
To ensure uniqueness, we split insertions into three stages (Figure 5). First, a thread reserves an empty bucket and publishes its attempt by storing the key it is inserting, along with an 'inserting' flag. Next, the thread checks the other positions in the probe sequence for that key, looking for other 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



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.

Fig. 4. Problems concurrently inserting keys

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.

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

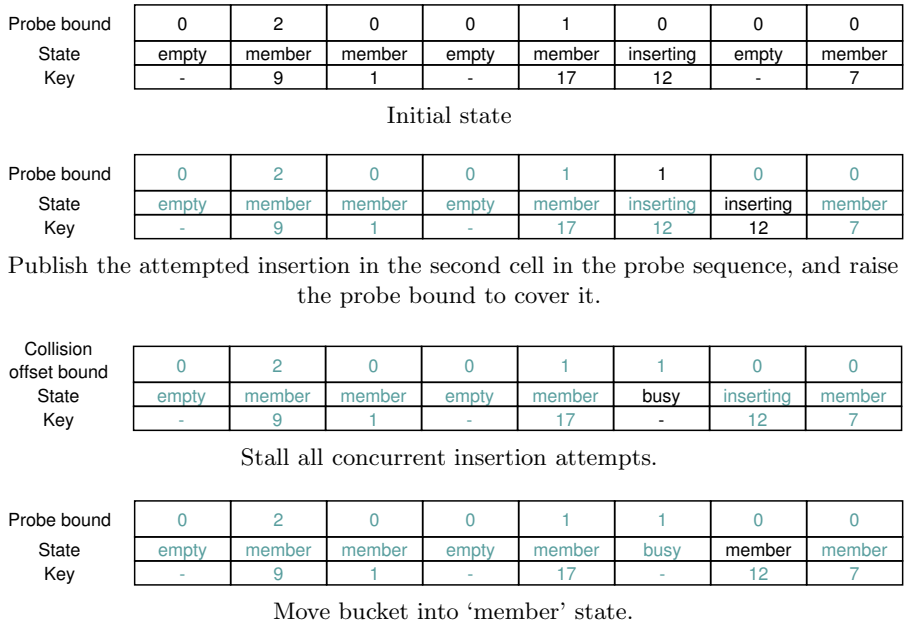


Fig. 5. Inserting key 12

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 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

```
23      word buckets[size] // ⟨key,state⟩
        word* Bucket(int h, int index): // Size must be a power of 2
25          return &buckets[(h + index*(index+1)/2) % size] // Quadratic probing
        bool DoesBucketContainCollision(int h, int index):
27          ⟨k,state⟩ := *Bucket(h,index)
            return (k ≠ - ∧ Hash(k) = h)
29  public:
        void Init():
31          for i := 0 .. size-1
                InitProbeBound(i)
33              buckets[i] := ⟨-,empty⟩
        bool Lookup(Key k): // Determine whether k is a member of the set
35          h := Hash(k)
            max := GetProbeBound(h)
37          for i := 0 .. max
                if *Bucket(h,i) = ⟨k,member⟩
39                  return true
            return false
41      bool Insert(Key k): // Insert k into the set if it is not a member
            h := Hash(k)
43          i := 0 // Reserve a cell
            while ¬CAS(Bucket(h,i), ⟨-,empty⟩, ⟨-,busy⟩)
45              i++
                if i ≥ size
47                  throw "Table full"
            do // Attempt to insert a unique copy of k
49              *Bucket(h,i) := ⟨k,inserting⟩
                ConditionallyRaiseBound(h,i)
51              max := GetProbeBound(h) // Scan through the probe sequence
                for j := 0 .. max
53                  if j ≠ i
                        if *Bucket(h,j) = ⟨k, inserting⟩ // Stall concurrent inserts
55                          CAS(Bucket(h,j), ⟨k,inserting⟩, ⟨-,busy⟩)
                        if *Bucket(h,j) = ⟨k,member⟩ // Abort if k already a member
57                          *Bucket(h,i) := ⟨-,busy⟩
                            ConditionallyLowerBound(h,i)
59                          *Bucket(h,i) := ⟨-,empty⟩
                            return false
61          while ¬CAS(Bucket(h,i), ⟨k,inserting⟩, ⟨k,member⟩)
            return true
63      bool Erase(Key k): // Remove k from the set if it is a member
            h := Hash(k)
65          max := GetProbeBound(h) // Scan through the probe sequence
            for i := 0 .. max
67              if *Bucket(h,i) = ⟨k,member⟩ // Remove a copy of ⟨k, member⟩
                    if CAS(Bucket(h,i), ⟨k,member⟩, ⟨-,busy⟩)
69                      ConditionallyLowerBound(h,i)
                        *Bucket(h,i) := ⟨-,empty⟩
71                      return true
            return false
73  }
```

Fig. 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.

Fig. 7. Problems assisting concurrent operations

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.

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

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

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

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

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

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

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

Fig. 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`.

Fig. 9. Inserting key 12 (lock-free algorithm)

```
    bool Insert(Key k): // Insert k into the set if it is not a member
65      h := Hash(k)
        i := -1 // Reserve a cell
67      do
            if ++i ≥ size
69              throw "Table full"
            ⟨version,state⟩ := Bucket(h,i)→vs
71      while ¬CAS(&Bucket(h,i)→vs, ⟨version,empty⟩, ⟨version,busy⟩)
        Bucket(h,i)→key := k
73      while true // Attempt to insert a unique copy of k
            *Bucket(h,i)→vs := ⟨version,visible⟩
75          ConditionallyRaiseBound(h,i)
            *Bucket(h,i)→vs := ⟨version,inserting⟩
77          r := Assist(k,h,i,version)
            if Bucket(h,i)→vs ≠ ⟨version,collided⟩
79              return true
            if ¬r
81              ConditionallyLowerBound(h,i)
                Bucket(h,i)→vs := ⟨version+1,empty⟩
83              return false
            version++
85  private:
    bool Assist(Key k,int h,int i,int ver_i): // Attempt to insert k at i
87      // Return true if no other cell seen in member state
        max := GetProbeBound(h) // Scan through probe sequence
89      for j := 0 .. max
            if i ≠ j
91              ⟨ver_j,state_j⟩ := Bucket(h,j)→vs
                if state_j = inserting ∧ Bucket(h,j)→key = k
93                  if j < i // Assist any insert found earlier in the probe sequence
                        if Bucket(h,j)→vs = ⟨ver_j,inserting⟩
95                          CAS(&Bucket(h,i)→vs, ⟨ver_i,inserting⟩, ⟨ver_i,collided⟩)
                            return Assist(k,h,j,ver_j)
97                  else // Fail any insert found later in the probe sequence
                        if Bucket(h,i)→vs = ⟨ver_i,inserting⟩
99                          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
101             if state_j = member ∧ Bucket(h,j)→key = k
                    if Bucket(h,j)→vs = ⟨ver_j,member⟩
103                     CAS(&Bucket(h,i)→vs,⟨ver_i,inserting⟩,⟨ver_i,collided⟩)
                        return false
105     CAS(&Bucket(h,i), ⟨ver_i,inserting⟩, ⟨ver_i,member⟩)
        return true
107 }
```

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

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 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.

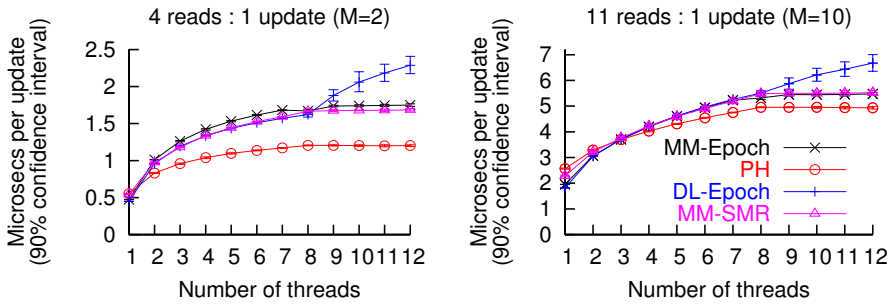


Fig. 11. Performance on 8-way SPARC machine

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 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.

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.