Lock-Free Resizeable Concurrent Tries

Aleksandar Prokopec, Phil Bagwell, Martin Odersky

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

Abstract. This paper describes an implementation of a non-blocking concurrent hash trie based on single-word compare-and-swap instructions in a shared-memory system. Insert, lookup and remove operations modifying different parts of the hash trie can be run completely independently. Remove operations ensure that the unneeded memory is freed and that the trie is kept compact. A pseudocode for these operations is presented and a proof of correctness is given – we show that the implementation is linearizable and lock-free. Finally, benchmarks are presented that compare concurrent hash trie operations against the corresponding operations on other concurrent data structures.

1 Introduction

In the presence of multiple processors data has to be accessed concurrently. Concurrent access to data requires synchronization in order to be correct. A traditional approach to synchronization is to use mutual exclusion locks. However, locks induce a performance degradation if a thread holding a lock gets delayed (e.g. by being preempted by the operating system). All other threads competing for the lock are prevented from making progress until the lock is released. More fundamentally, mutual exclusion locks are not fault tolerant – a failure may prevent progress indefinitely.

A lock-free concurrent object guarantees that if several threads attempt to perform an operation on the object, then at least some thread will complete the operation after a finite number of steps. Lock-free data structures are in general more robust than their lock-based counterparts [10], as they are immune to deadlocks, and unaffected by thread delays and failures. Universal methodologies for constructing lock-free data structures exist [9], but they serve as a theoretical foundation and are in general too inefficient to be practical – developing efficient lock-free data structures still seems to necessitate a manual approach.

Trie is a data structure with a wide range of applications first developed by Brandais [6] and Fredkin [7]. Hash array mapped tries described by Bagwell [1] are a specific type of tries used to store key-value pairs. The search for the key is guided by the bits in the hashcode value of the key. Each hash trie node stores references to subtries inside an array, which is indexed with a bitmap. This makes hash array mapped tries both space-efficient and cache-aware – the bitmap and the array can be stored within the same cache line. A similar approach was taken in the dynamic array data structures [8]. Hash array mapped tries are space-efficient and ensure that they are compressed as elements are being removed. They are well-suited for applications where the size bounds of the data structure are not known in advance and vary through time. In this paper we describe in detail a non-blocking implementation of the hash array mapped trie.

Our contributions are the following:

1. We introduce a completely lock-free concurrent hash trie data structure for a shared-memory system based on single-word compare-and-swap instructions. A complete pseudocode is included in the paper.
2. Our implementation maintains the space-efficiency of sequential hash tries. Additionally, remove operations check to see if the concurrent hash trie can be contracted after a key has been removed, thus saving space and ensuring that the depth of the trie is optimal.
3. There is no stop-the-world dynamic resizing phase during which no operation can be completed – the data structure grows with each subsequent insertion and removal. This makes our data structure suitable for real-time applications.
4. We present a proof of correctness and show that all operations are linearizable and lock-free.
5. We present benchmarks that compare performance of concurrent hash tries against other concurrent data structures. We interpret the results.

The rest of the paper is organized as follows. Section 2 describes sequential hash tries and several attempts to make their operations concurrent. It then presents case studies with concurrent hash trie operations. Section 3 presents the algorithm for concurrent hash trie operations and describes it in detail. Section 4 presents the outline of the correctness proof – a complete proof is given in the appendix. Section 5 contains experimental results and their interpretation. Section 6 presents related work and section 7 concludes.

2 Discussion

Hash array mapped tries (from now on hash tries) described previously by Bagwell [1] are trees that have 2 types of nodes – internal nodes and leaves. Leaves store key-value bindings. Internal nodes have a 2^W-way branching factor. In a straightforward implementation, each internal node is a 2^W-element array. Finding a key proceeds in the following manner. If the internal node is at the root, the initial W bits of the key hashcode are used as an index in the array. If the internal node is at the level l, then W bits of the hashcode starting from the position $W * l$ are used. This is repeated until a leaf or an empty entry is found. Insertion and removal are similar.

Such an implementation is space-inefficient – most entries in the internal nodes are never used. To ensure space efficiency, each internal node contains a bitmap of length 2^W. If a bit is set, then its corresponding array entry contains an element. The corresponding entry for a bit on position i in the bitmap bmp is calculated as $\#((i-1) \odot bmp)$, where $\#$ is the bitcount and $\odot$ is a logical AND operation. The W bits of the hashcode relevant at some level l are used to compute the index i as before. At all times an invariant is preserved that the bitmap bitcount is equal to the array length. Typically, W is 5 since that ensures that 32-bit integers can be used as bitmaps. An example hash trie is shown in Fig. 1A. The used space is $O(n)$. The expected depth is logarithmic in the number of elements added – this drives the running time of operations.

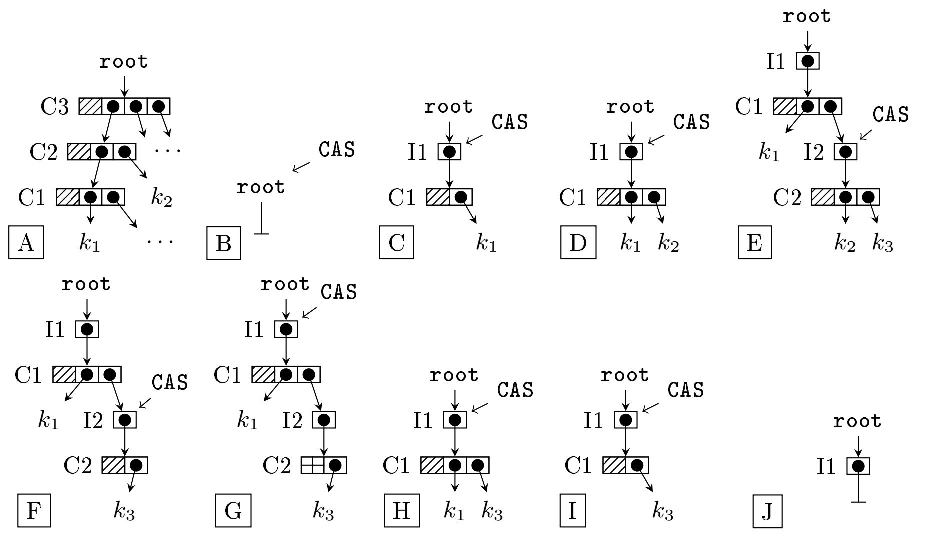

Fig. 1. Hash trie and Ctrie examples

We want to preserve the nice properties of hash tries – space-efficiency, cache-awareness and the expected depth of $O(\log_{2^W}(n))$, where n is the number of elements stored in the trie and 2^W is the bitmap length. We also want to make hash tries a concurrent data structure that can be accessed by multiple threads. In doing so, we avoid locks and rely solely on CAS instructions. Furthermore, we ensure that the new data structure has the lock-freedom property. We call the data structure a *Ctrie*. In the remainder of this chapter we give several examples.

Assume that we have a hash trie from Fig. 1A and that a thread T_1 decides to insert a new key below the node `C1`. One way to do this is to do a CAS on the bitmap in `C1` to set the bit that corresponds to the new entry in the array, and then CAS the entry in the array to point to the new key. This requires all the arrays to have additional empty entries, leading to inefficiencies. A possible solution is to keep a pointer to the array inside `C1` and do a CAS on that pointer with the updated copy of the array. The fundamental problem that still remains is that such an insertion does not happen atomically. It is possible that some other thread T_2 also tries to insert below `C1` after its bitmap is updated, but before the array pointer is updated. Lock-freedom is not ensured if T_2 were to wait for T_1 to complete.

Another solution is for T_1 to create an updated version of `C1` called `C1'` with the updated bitmap and the new key entry in the array, and then do a CAS in the entry within the `C2` array that points to `C1`. The change is then done atomically. However, this approach does not work. Assume that another thread T_2 decides to insert a key below the node `C2` at the time when T_1 is creating `C1'`. To do this, it has to read `C2` and create its updated copy `C2'`. Assume that after that, T_1 does the CAS in `C2`. The copy `C2'` will not reflect the changes by T_1. Once T_2 does a CAS in the `C3` array, the key inserted by T_1 is lost.

To solve this problem we define a new type of a node that we call an *indirection node*. This node remains present within the Ctrie even if nodes above and below it change. We now show an example of a sequence of Ctrie operations.

Every Ctrie is defined by the `root` reference (Fig. 1B). Initially, the `root` is set to a special value called `null`. In this state the Ctrie corresponds to an empty set, so all lookups fail to find a value for any given key and all remove operations fail to remove a binding.

Assume that a key k_1 has to be inserted. First, a new node `C1` of type `CNode` is created, so that it contains a single key `k1` according to hash trie invariants. After that, a new node `I1` of type `INode` is created. The node `I1` has a single field *main* (Fig. 2) that is initialized to `C1`. A CAS instruction is then performed at the `root` reference (Fig. 1B), with the expected value `null` and the new value `I1`. If a CAS is successful, the insertion is completed and the Ctrie is in a state shown in Fig. 1C. Otherwise, the insertion must be repeated.

Assume next that a key k_2 is inserted such that its hashcode prefix is different from that of k_1. By the hash trie invariants, k_2 should be next to k_1 in `C1`. The thread that does the insertion first creates an updated version of `C1` and then does a CAS at the `I1.main` (Fig. 1C) with the expected value of `C1` and the updated node as the new value. Again, if the CAS is not successful, the insertion process is repeated. The Ctrie is now in the state shown in Fig. 1D.

If some thread inserts a key k_3 with the same initial bits as k_2, the hash trie has to be extended with an additional level. The thread starts by creating a new node `C2` of type `CNode` containing both k_2 and k_3. It then creates a new node `I2` and sets `I2.main` to `C2`. Finally, it creates a new updated version of `C1` such that it points to the node `I2` instead of the key k_2 and does a CAS at `I1.main` (Fig. 1D). We obtain a Ctrie shown in Fig. 1E.

Assume now that a thread T_1 decides to remove k_2 from the Ctrie. It creates a new node `C2'` from `C2` that omits the key k_2. It then does a CAS on `I2.main` to set it to `C2'` (Fig. 1E). As before, if the CAS is not successful, the operation is restarted. Otherwise, k_2 will no longer be in the trie – concurrent operations will only see k_1 and k_3 in the trie, as shown in Fig. 1F. However, the key k_3 could be moved further to the root - instead of being below the node `C2`, it could be directly below the node `C1`. In general, we want to ensure that the path from the root to a key is as short as possible. If we do not do this, we may end up with a lot of wasted space and an increased depth of the Ctrie.

For this reason, after having removed a key, a thread will attempt to contract the trie as much as possible. The thread T_1 that removed the key has to check whether or not there are less than 2 keys remaining within `C2`. There is only a single key, so it can create a copy of `C1` such that the key k_3 appears in place of the node `I2` and then do a CAS at `I1.main` (Fig. 1F). However, this approach does not work. Assume there was another thread T_2 that decides to insert a new key below the node `I2` just before T_1 does the CAS at `I1.main`. The key inserted by T_2 is lost as soon as the CAS at `I1.main` occurs.

To solve this, we relax the invariants of the data structure. We introduce a new type of a node - a tomb node. A tomb node is simply a node that holds a single key. No thread may modify a node of type `INode` if it contains a tomb node. In our example, instead of directly modifying `I1`, thread T_1 must first create a tomb node that contains the key k_3. It then does a CAS at `I2.main` to set it to the tomb node. After having done this (Fig. 1G), T_1 may create a contracted version of `C1` and do a CAS at `I1.main`, at that point we end up with a trie of an optimal size (Fig. 1H). If some other thread T_2 attempts to modify `I2` after it has been *tombed*, then it must first do the same thing T_1 is attempting to do - move the key k_3 back below `C2`, and only then proceed with its original operation. We call an `INode` that points to a tomb node a *tomb-I-node*. We say that a tomb-I-node in the example above is *resurrected*.

If some thread decides to remove k_1, it proceeds as before. However, even though k_3 now remains the only key in `C1` (Fig. 1I), it does not get tombed. The reason for this is that we treat nodes directly below the root differently. If k_3 were next removed, the trie would end up in a state shown in Fig. 1J, with the `I1.main` set to `null`. We call this type of an `INode` a *null-I-node*.

```
                         MainNode: CNode | SNode
root: INode                                          structure SNode {
                         structure CNode {             k: KeyType
structure INode {          bmp: integer                v: ValueType
  main: MainNode           array: Array[2^W]           tomb: boolean
}                        }                           }
```

Fig. 2. Types and data structures

3 Algorithm

We present the pseudocode of the algorithm in figures 3, 4 and 5. The pseudocode assumes C-like semantics of conditions in *if* statements – if the first condition in a conjunction fails, the second one is never evaluated. The pseudocode contains pattern matching constructs used to match a node against its type. All occurences of pattern matching can be replaced with a sequence of *if-then-else* statements – we use pattern matching for conciseness. The colon (`:`) in the pattern matching cases is read as *has type*. The keyword `def` denotes a procedure definition. Reads and CAS instructions written in capitals are atomic – they

```
 1 def insert(k, v)
 2   r = READ(root)
 3   if r = null ∨ isNullInode(r) {
 4     scn = CNode(SNode(k, v, ⊥))
 5     nr = INode(scn)
 6     if !CAS(root, r, nr) insert(k, v)
 7   } else if ¬iinsert(r, k, v, 0, null)
 8     insert(k, v)
 9
10 def remove(k)
11   r = READ(root)
12   if r = null return NOTFOUND
13   else if isNullInode(r) {
14     CAS(root, r, null)
15     return remove(k)
16   } else {
17     res = iremove(r, k, 0, null)
18     if res ≠ RESTART return res
19     else remove(k)
20   }
21
22 def lookup(k)
23   r = READ(root)
24   if r = null return NOTFOUND
25   else if isNullInode(r) {
26     CAS(root, r, null)
27     return lookup(k)
28   } else {
29     res = ilookup(r, k, 0, null)
30     if res ≠ RESTART return res
31     else return lookup(k)
32   }
33
34 def ilookup(i, k, lev, parent)
35   READ(i.main) match {
36   case cn: CNode =>
37     flag, pos = flagpos(k.hc, lev, cn.bmp)
38     if cn.bmp ⊙ flag = 0 return NOTFOUND
39     cn.array(pos) match {
40     case sin: INode =>
41       return ilookup(sin, k, lev + W, i)
42     case sn: SNode ∧ ¬sn.tomb =>
43       if sn.k = k return sn.v
44       else return NOTFOUND
45     }
46   case (sn: SNode ∧ sn.tomb) ∨ null =>
47     if parent ≠ null clean(parent)
48     return RESTART
49   }
```

Fig. 3. Basic operations I

occur at one point in time. This high level pseudocode may not be optimal in all cases – the source code contains a more efficient implementation [1].

Operations start by reading the `root` (lines 2, 11 and 23). If the `root` is `null` then the trie is empty, so neither removal nor lookup finds a key. If the `root` points to an `INode` that is set to `null` (as in Fig. 1J), then the root is set back to just `null` before repeating. In both the previous cases, an insertion will replace the `root` reference with a new `CNode` with the appropriate key.

If the `root` is neither `null` nor a null-I-node then the node below the root I-node is read (lines 35, 51 and 79), and we proceed casewise. If the node pointed at by the I-node is a `CNode`, an appropriate entry in its array must be found. The method `flagpos` computes the values `flag` and `pos` from the hashcode hc of the key, bitmap bmp of the cnode and the current level lev. The relevant `flag` in the bitmap is defined as $(hc >> (k \cdot lev)) \odot ((1 << k) - 1)$, where 2^k is the length of the bitmap. The position `pos` within the array is given by the expression $\#((flag - 1) \odot bmp)$, where $\#$ is the bitcount. The `flag` is used to check if the appropriate branch is in the `CNode` (lines 38, 54, 82). If it is not, lookups and removes end, since the desired key is not in the Ctrie. An insert creates an updated copy of the current `CNode` with the new key. If the branch is in the trie, `pos` is used as an index into the array. If an I-node is found, we repeat the operation recursively. If a key-value binding (an `SNode`) is found, then a lookup compares the keys and returns the binding if they are the same. An insert operation will either replace the old binding if the keys are the same, or

[1] See: http://github.com/axel22/Ctries

```
50 def iinsert(i, k, v, lev, parent)
51   READ(i.main) match {
52   case cn: CNode =>
53     flag, pos = flagpos(k.hc, lev, cn.bmp)
54     if cn.bmp ⊙ flag = 0 {
55       nsn = SNode(k, v, ⊥)
56       narr = cn.array.inserted(pos, nsn)
57       ncn = CNode(narr, bmp | flag)
58       return CAS(i.main, cn, ncn)
59     }
60     cn.array(pos) match {
61     case sin: INode =>
62       return iinsert(sin, k, v, lev + W, i)
63     case sn: SNode ∧ ¬sn.tomb =>
64       nsn = SNode(k, v, ⊥)
65       if sn.k = k {
66         ncn = cn.updated(pos, nsn)
67         return CAS(i.main, cn, ncn)
68       } else {
69         nin = INode(CNode(sn, nsn, lev + W))
70         ncn = cn.updated(pos, nin)
71         return CAS(i.main, cn, ncn)
72       }
73     }
74   case (sn: SNode ∧ sn.tomb) ∨ null =>
75     if parent ≠ null clean(parent)
76     return ⊥
77   }
```

```
78 def iremove(i, k, lev, parent)
79   READ(i.main) match {
80   case cn: CNode =>
81     flag, pos = flagpos(k.hc, lev, cn.bmp)
82     if cn.bmp ⊙ flag = 0 return NOTFOUND
83     res = cn.array(pos) match {
84     case sin: INode =>
85       return iremove(sin, k, lev + W, i)
86     case sn: SNode ∧ ¬sn.tomb =>
87       if sn.k = k {
88         narr = cn.array.removed(pos)
89         ncn = CNode(narr, bmp ^ flag)
90         if cn.array.length = 1 ncn = null
91         if CAS(i.main, cn, ncn) return sn.v
92         else return RESTART
93       } else return NOTFOUND
94     }
95     if res = NOTFOUND ∨ res = RESTART return res
96     if parent ne null ∧ tombCompress()
97       contractParent(parent, in, k.hc, lev - W)
98   case (sn: SNode ∧ sn.tomb) ∨ null =>
99     if parent ≠ null clean(parent)
100    return RESTART
101  }
```

Fig. 4. Basic operations II

otherwise extend the trie below the `CNode`. A remove compares the keys – if they are the same it replaces the `CNode` with its updated version without the key.

After a key was removed, the trie must be contracted. A remove first attempts to create a tomb from the current `CNode`. It reads the node below the current I-node to check if it is still a `CNode`. It then calls `toWeakTombed` that creates a *weak tomb* from the given `CNode`. A weak tomb is defined as follows. If the number of nodes below the `CNode` that are not null-I-nodes is greater than 1, then it is the `CNode` itself – we say that there is nothing to entomb. If the number of such nodes is 0, then the weak tomb is `null`. Otherwise, if the single branch below the `CNode` is a key-value binding or a tomb-I-node (also called a *singleton*), the weak tomb is the tomb node with that binding. If the single branch is another `CNode`, a weak tomb is a copy of the current `CNode` without the null-I-nodes.

The procedure `tombCompress` continually tries to entomb the current `CNode` until it finds out that there is nothing to entomb or it succeeds. The CAS in line 132 corresponds to the one in Fig. 1F. If it succeeds and the weak tomb was either a `null` or a tomb node, it will return `true`, meaning that the parent node should be contracted. The contraction is done in `contractParent`, that checks if the I-node is still reachable from its parent and then contracts the `CNode` below the parent - it removes the null-I-node (line 148) or resurrects a tomb-I-node into an `SNode` (line 152). The latter corresponds to the CAS in Fig. 1G.

If any operation encounters a `null` or a tomb node, it attempts to fix the Ctrie before proceeding, since the Ctrie is in a *relaxed* state. A tomb node may

```
def toCompressed(cn)
  num = bit#(cn.bmp)
  if num = 1 $\wedge$ isTombInode(cn.array(0))
    return cn.array(0).main
  ncn = cn.filtered(_.main $\neq$ null)
  rarr = ncn.array.map(resurrect(_))
  if bit#(ncn.bmp) > 0
    return CNode(rarr, ncn.bmp)
  else return null

def toWeakTombed(cn)
  farr = cn.array.filtered(_.main $\neq$ null)
  nbmp = cn.bmp.filtered(_.main $\neq$ null)
  if farr.length > 1 return cn
  if farr.length = 1
    if isSingleton(farr(0))
      return farr(0).tombed
    else CNode(farr, nbmp)
  return null

def clean(i)
  m = READ(i.main)
  if m $\in$ CNode
    CAS(i.main, m, toCompressed(m))

def tombCompress(i)
  m = READ(i.main)
  if m $\notin$ CNode return $\bot$
  mwt = toWeakTombed(m)
  if m = mwt return $\bot$
  if CAS(i.main, m, mwt) mwt match {
    case null $\vee$ (sn: SNode $\wedge$ sn.tomb) =>
      return $\top$
    case _ => return $\bot$
  } else return tombCompress()

def contractParent(parent, i, hc, lev)
  m, pm = READ(i.main), READ(parent.main)
  pm match {
  case cn: CNode =>
    flag, pos = flagpos(k.hc, lev, cn.bmp)
    if bmp $\odot$ flag = 0 return
    sub = cn.array(pos)
    if sub $\neq$ i return
    if m = null {
      ncn = cn.removed(pos)
      if !CAS(parent.main, cn, ncn)
        contractParent(parent, i, hc, lev)
    } else if isSingleton(m) {
      ncn = cn.updated(pos, m.untombed)
      if !CAS(parent.main, cn, ncn)
        contractParent(parent, i, hc, lev)
    }
  case _ => return
  }
```

Fig. 5. Compression operations

have originated from a remove operation that will attempt to contract the tomb node at some time in the future. Rather than waiting for that remove to do its work, the current operation contracts the tomb itself. It will invoke the `clean` operation on the parent I-node, which will attempt to exchange the `CNode` below the parent I-node with its compression. A `CNode` compression is defined as follows – if the `CNode` has a single tomb node directly beneath, then it is that tomb node. Otherwise, the compression is the copy of the `CNode` without the null-I-nodes (the `filtered` call in the `toCompressed` procedure) and with all the tomb-I-nodes resurrected to regular key nodes (this is what the `map` and `resurrect` calls do). Going back to our previous example, if in Fig. 1G some other thread were to attempt to write to `I2`, it would first do a `clean` operation on the parent `I1` of `I2` – it would contract the trie in the same way as the remove would have.

4 Correctness

As illustrated by the examples in the previous section, designing a correct lock-free algorithm is not straightforward. One of the reasons for this is that all possible interleavings of steps of different threads executing the operations have to be considered. For brevity, this section gives only the outline of the correctness proof. There are three main criteria for correctness. *Safety* means that the Ctrie corresponds to some abstract set of keys and that all operations change the corresponding abstract set of keys consistently. An operation is *linearizable* if any external observer can only observe the operation as if it took place instantaneously at some point between its invocation and completion [9] [11]. *Lock-freedom* means that if some number of threads execute operations concurrently, then after a finite number of steps some operation must complete [9].

We assume that the Ctrie has a branching factor 2^W. Each node in the Ctrie is identified by its type, level in the Ctrie l and the hashcode prefix p. The hashcode prefix is the sequence of branch indices that have to be followed from the root in order to reach the node. For a C-node $cn_{l,p}$ and a key k with the hashcode $h = r_0 \cdot r_1 \cdots r_n$, we denote $cn.sub(k)$ as the branch with the index r_l or *null* if such a branch does not exist. We define the following invariants:

INV1 For every I-node $in_{l,p}$, $in_{l,p}.main$ is a C-node $cn_{l,p}$, a tombed S-node $sn\dagger$ or *null*.
INV2 For every C-node the length of the array is equal to the bitcount in the bitmap.
INV3 If a flag i in the bitmap of $cn_{l,p}$ is set, then corresponding array entry contains an I-node $in_{l+W,p\cdot r}$ or an S-node.
INV4 If an entry in the array in $cn_{l,p}$ contains an S-node sn, then p is the prefix of the hashcode $sn.k$.
INV5 If an I-node $in_{l,p}$ contains an S-node sn, then p is the prefix of the hashcode $sn.k$.

We say that the Ctrie is *valid* if and only if the invariants hold. The relation $hasKey(node, x)$ holds if and only if the key x is within an S-node reachable from *node*. A valid Ctrie is *consistent* with an abstract set $\mathbb{A}$ if and only if $\forall k \in \mathbb{A}$ the relation $hasKey(root, k)$ holds and $\forall k \notin \mathbb{A}$ it does not. A Ctrie lookup is *consistent* with the abstract set semantics if and only if it finds the keys in the abstract set and does not find other keys. A Ctrie insertion or removal is *consistent* with the abstract set semantics if and only if it produces a new Ctrie consistent with a new abstract set with or without the given key, respectively.

Lemma 1. *If an I-node in is either a null-I-node or a tomb-I-node at some time* t_0 *then* $\forall t > t_0$ *in.main is never written to. We refer to such I-nodes as nonlive.*

Lemma 2. *C-nodes and S-nodes are immutable – once created, they do not change the value of their fields.*

Lemma 3. *Invariants INV1-3 always hold.*

Lemma 4. *If a CAS instruction makes an I-node in unreachable from its parent at some time* t_0*, then in is nonlive at* t_0*.*

Lemma 5. *Reading a cn such that* $cn.sub(k) = sn$ *and* $k = sn.k$ *at some time* t_0 *means that* $hasKey(root, k)$ *holds at* t_0*.*

For a given Ctrie, we say that the longest path for a hashcode $h = r_0 \cdot r_1 \cdots r_n$, $length(r_i) = W$, is the path from the root to a leaf such that at each C-node $cn_{i,p}$ the branch with the index r_i is taken.

Lemma 6. *Assume that the Ctrie is an valid state. Then every longest path ends with an S-node, C-node or null.*

Lemma 7. *Assume that a C-node cn is read from $in_{l,p}.main$ at some time t_0 while searching for a key k. If $cn.sub(k) = null$ then $hasKey(root, k)$ is not in the Ctrie at t_0.*

Lemma 8. *Assume that the algorithm is searching for a key k and that an S-node sn is read from $cn.array(i)$ at some time t_0 such that $sn.k \neq k$. Then the relation $hasKey(root, k)$ does not hold at t_0.*

Lemma 9. *1. Assume that one of the CAS in lines 58 and 71 succeeds at time t_1 after in.main was read in line 51 at time t_0. Then $\forall t, t_0 \leq t < t_1$, relation $hasKey(root, k)$ does not hold.*

2. Assume that the CAS in lines 67 succeeds at time t_1 after in.main was read in line 51 at time t_0. Then $\forall t, t_0 \leq t < t_1$, relation $hasKey(root, k)$ holds.

3. Assume that the CAS in line 91 succeeds at time t_1 after in.main was read in line 79 at time t_0. Then $\forall t, t_0 \leq t < t_1$, relation $hasKey(root, k)$ holds.

Lemma 10. *Assume that the Ctrie is valid and consistent with some abstract set $\mathbb{A}$ $\forall t, t_1 - \delta < t < t_1$. CAS instructions from lemma 9 induce a change into a valid state that is consistent with the abstract set semantics.*

Lemma 11. *Assume that the Ctrie is valid and consistent with some abstract set $\mathbb{A}$ $\forall t, t_1 - \delta < t < t_1$. If one of the operations clean, tombCompress or contractParent succeeds with a CAS at t_1, the Ctrie will remain valid and consistent with the abstract set $\mathbb{A}$ at t_1.*

Corollary 1. *Invariants INV4,5 always hold due to lemmas 10 and 11.*

Theorem 1 (Safety). *At all times t, a Ctrie is in a valid state $\mathbb{S}$, consistent with some abstract set $\mathbb{A}$. All Ctrie operations are consistent with the semantics of the abstract set $\mathbb{A}$.*

Theorem 2 (Linearizability). *Ctrie operations are linearizable.*

Lemma 12. *If a CAS that does not cause a consistency change in one of the lines 58, 67, 71, 125, 132, 148 or 152 fails at some time t_1, then there has been a state (configuration) change since the time t_0 when a respective read in one of the lines 51, 51, 51, 123, 128, 139 or 139 occured.*

Lemma 13. *In each operation there is a finite number of execution steps between consecutive CAS instructions.*

Corollary 2. *There is a finite number of execution steps between two state changes. This does not imply that there is a finite number of execution steps between two operations. A state change is not necessarily a consistency change.*

We define the **total path length** d as the sum of the lengths of all the paths from the root to some leaf. Assume the Ctrie is in a valid state. Let n be the number of reachable null-I-nodes in this state, t the number of reachable tomb-I-nodes, l the number of live I-nodes, r the number of single tips of any length and d the total path length. We denote the state of the Ctrie as $\mathbb{S}_{n,t,l,r,d}$. We call the state $\mathbb{S}_{0,0,l,r,d}$ the **clean** state.

Lemma 14. *Observe all CAS instructions that never cause a consistency change and assume they are successful. Assuming there was no state change since reading in prior to calling clean, the CAS in line 125 changes the state of the Ctrie from the state* $\mathbb{S}_{n,t,l,r,d}$ *to either* $\mathbb{S}_{n+j,t,l,r-1,d-1}$ *where* $r > 0$, $j \in \{0, 1\}$ *and* $d \geq 1$, *or to* $\mathbb{S}_{n-k,t-j,l,r,d' \leq d}$ *where* $k \geq 0$, $j \geq 0$, $k + j > 0$, $n \geq k$ *and* $t \geq j$. *Furthermore, the CAS in line 14 changes the state of the Ctrie from* $\mathbb{S}_{1,0,0,0,1}$ *to* $\mathbb{S}_{0,0,0,0,0}$. *The CAS in line 26 changes the state from* $\mathbb{S}_{1,0,0,0,1}$ *to* $\mathbb{S}_{0,0,0,0,0}$. *The CAS in line 132 changes the state from* $\mathbb{S}_{n,t,l,r,d}$ *to either* $\mathbb{S}_{n+j,t,l,r-1,d-j}$ *where* $r > 0$, $j \in \{0, 1\}$ *and* $d \geq j$, *or to* $\mathbb{S}_{n-k,t,l,r,d' \leq d}$ *where* $k > 0$ *and* $n \geq k$. *The CAS in line 148 changes the state from* $\mathbb{S}_{n,t,l,r,d}$ *to* $\mathbb{S}_{n-1,t,l,r+j,d-1}$ *where* $n > 0$ *and* $j \geq 0$. *The CAS in line 152 changes the state from* $\mathbb{S}_{n,t,l,r}$ *to* $\mathbb{S}_{n,t-1,l,r+j,d-1}$ *where* $j \geq 0$.

Lemma 15. *If the Ctrie is in a clean state and* n *threads are executing operations on it, then some thread will execute a successful CAS resulting in a consistency change after a finite number of execution steps.*

Theorem 3 (Lock-freedom). *Ctrie operations are lock-free.*

5 Experiments

We show benchmark results in Fig. 6. All the measurements were performed on a quad-core 2.67 GHz i7 processor with hyperthreading. We followed established performance measurement methodologies [2]. We compare the performance of Ctries against that of `ConcurrentHashMap` and `ConcurrentSkipListMap` [3] [4] data structures from the Java standard library.

In the first experiment, we insert a total of N elements into the data structures. The insertion is divided equally among P threads, where P ranges from 1 to 8. The results are shown in Fig. 6A-D. Ctries outperform concurrent skip lists for $P = 1$ (Fig. 6A). We argue that this is due to a fewer number of indirections in the Ctrie data structure. A concurrent skip list roughly corresponds to a balanced binary tree that has a branching factor 2. Ctries normally have a branching factor 32, thus having a much lower depth. A lower depth means less indirections and consequently fewer cache misses when searching the Ctrie.

We can also see that the Ctrie sometimes outperforms a concurrent hash table for $P = 1$. The reason is that the hash table has a fixed size and is resized once the load factor is reached – roughly speaking, a new table has to be allocated and all the elements from the previous hash table have to be copied into the new hash table. To do this, parts of the hash table have to be locked – other threads adding new elements into the table have to wait until the resize completes. This problem is much more apparent in Fig. 6B where $P = 8$. Fig. 6C,D show how the insertion scales for the number of elements $N = 200k$ and $N = 1M$, respectively. Due to the use of hyperthreading on the i7, we do not get significant speedups when $P > 4$ for these data structures. We next measure the performance for the remove operation (Fig. 6E-H). Each data structure starts with N elements and then emptied concurrently by P threads. The keys being removed are divided equally among the threads. For $P = 1$ Ctries are clearly outperformed by both other data structures. However, it should be noted that concurrent hash table does not shrink once the number of keys becomes much lower than the table size. This is space-inefficient – a hash table contains many elements at some point during the runtime of the application will continue to use the memory it does not need until the application ends. The slower Ctrie performance seen in Fig. 6E for $P = 1$ is attributed to the additional work the remove operation does to keep the Ctrie compact. However, Fig. 6F shows that the Ctrie remove operation scales well for $P = 8$, as it outperforms both skip list and hash table removals. This is also apparent in Fig. 6G,H. In the next experiment, we populate all the data structures with N elements and then do a lookup for every element once. The set of elements to be looked up is divided equally among P threads. From Fig. 6I-L it is apparent that concurrent hash tables have a much more efficient lookups than other data structures. This is not surprising since they are a flat data structure – a lookup typically consists of a single read in the table, possibly followed by traversing the collision chain within the bucket. Although a Ctrie lookup outperforms a concurrent skip list when $P = 8$, it still has to traverse more indirections than a hash table. Finally, we do a series of benchmarks with both lookups and insertions to determine the percentage of lookups for which the concurrent hash table performance equals that of concurrent tries. Our test inserts new elements into the data structures using P threads. A total of N elements are inserted. After each insert, a lookup for a random element is performed r times, where r is the ratio of lookups per insertion. Concurrent skip lists scaled well in these tests but had low absolute performance, so they are excluded from the graphs for clarity. When using $P = 2$ threads, the ratio where the running time is equal for both concurrent hash tables and concurrent tries is $r = 2$. When using $P = 4$ threads this ratio is $r = 5$ and for $P = 8$ the ratio is $r = 9$. As the number of threads increases, more opportunity for parallelism is lost during the resize phase in concurrent hash tables, hence the ratio increases. This is shown in Fig. 6M-O. In the last benchmark (Fig. 6P) we preallocate the array for the concurrent hash table to avoid resize phases – in this case the hash table outperforms the concurrent trie. The performance gap decreases as the number of threads approaches $P = 8$. The downside is that a large amount of memory has to be used for the hash table and the size needs to be known in advance.

6 Related work

Concurrent programming techniques and important results in the area are covered by Shavit and Herlihy [9]. An overview of concurrent data structures is given by Moir and Shavit [10]. There is a body of research available focusing on concurrent lists, queues and concurrent priority queues [5] [10]. While linked lists are inefficient as sets or maps because they do not scale well, the latter two do not support the basic operations on sets and maps, so we exclude these from the further discussion and focus on more suitable data structures.

Hash tables are typically resizeable arrays of buckets. Each bucket holds some number of elements that is expected to be constant. The constant number of elements per bucket necessitates resizing the data structure. Sequential hash tables amortize the cost of resizing the table over other operations [14], achieving constant-time operations. While the individual concurrent hash table operations such as insertion or removal can be performed in a lock-free manner as shown by Maged [4], resizing is typically implemented with a global lock. Although the cost of resize is amortized against operations by one thread, this approach does not guarantee horizontal scalability. Lea developed an extensible hash algorithm that allows concurrent searches during the resizing phase, but not concurrent insertions and removals [3]. Shalev and Shavit propose split-ordered lists which keep a table of hints into a linked list in a way that does not require rearranging the elements of the linked list when resizing [15]. This approach is quite innovative, but it is unclear how to shrink the hint table if most of the keys are removed, while preserving lock-freedom.

Skip lists are a data structure that stores elements in a linked list. There are multiple levels of linked lists that allow logarithmic-time insertions, removals and lookups. Skip lists were originally invented by Pugh [16]. Pugh proposed concurrent skip lists which achieve synchronization through the use of locks [17]. Concurrent non-blocking skip lists were later implemented by Lev, Herlihy, Luchangco and Shavit [18] and Lea [3]. Concurrent binary search trees were proposed by Kung and Lehman [19] – their implementation uses a constant number of locks at a time that exclude other insertion and removal operations, while lookups can proceed concurrently. Bronson et al. presented a scalable concurrent implementation of an AVL tree based on transactional memory mechanisms that require a fixed number of locks to perform deletions [20]. Recently, the first non-blocking implementation of a binary search tree was proposed [21].

Tries were originally proposed by Brandais [6] and Fredkin [7]. Trie hashing was applied to accessing files stored on the disk by Litwin [12]. Litwin, Sagiv and Vidyasankar implemented trie hashing in a concurrent setting [13], however, they did so by using mutual exclusion locks. Hash array mapped trees, or hash tries, are tries for shared-memory proposed by Bagwell [1]. To our knowledge, there is no nonblocking concurrent implementation of hash tries prior our work.

7 Conclusion

We described a lock-free concurrent implementation of the hash trie data structure. Our implementation supports insertion, remove and lookup operations. It is space-efficient in the sense that it keeps a minimal amount of information in the internal nodes. It is compact in the sense that after all removal operations complete, all paths from the root to a leaf containing a key are as short as possible. Operations are worst-case logarithmic with a low constant factor ($O(\log_{32} n)$). Its performance is comparable to that of the similar concurrent data structures. The data structure grows dynamically – it uses no locks and there is no resizing phase. We proved that it is linearizable and lock-free.

In the future we plan to extend the algorithm with operations like *move key*, that reassigns a value from one key to another atomically. One research direction is supporting efficient aggregation operations on the keys stored in the Ctrie. One example of such an operation is the size of the Ctrie. Finally, we plan to develop an efficient lock-free snapshot operation for the concurrent trie that allows traversal of all the keys present in the data structure at the time at which the snapshot was created. One possible approach to doing so is to, roughly speaking, keep a partial history in the indirection nodes.

References

1. P. Bagwell: Ideal Hash Trees. 2002.
2. A. Georges, D. Buytaert, L. Eeckhout: Statistically Rigorous Java Performance Evaluation. OOPSLA, 2007.
3. Doug Lea's Home Page: http://gee.cs.oswego.edu/
4. Maged M. Michael: High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. SPAA, 2002.
5. Timothy L. Harris: A Pragmatic Implementation of Non-Blocking Linked-Lists. IEEE Symposium on Distributed Computing, 2001.
6. R. Brandais: File searching using variable length keys. Proceedings of Western Joint Computer Conference, 1959.
7. E. Fredkin: Trie memory. Communications of the ACM, 1960.
8. A. Silverstein: Judy IV Shop Manual. 2002.
9. N. Shavit, M. Herlihy: The Art of Multiprocessor Programming. Morgan Kaufmann, 2008.
10. M. Moir, N. Shavit: Concurrent data structures. Handbook of Data Structures and Applications, Chapman and Hall, 2004.
11. M. Herlihy, J. Wing: Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems, 1990.
12. W. Litwin: Trie Hashing. ACM, 1981.
13. W. Litwin, Y. Sagiv, K. Vidyasankar: Concurrency and Trie Hashing. ACM, 1981.
14. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms, 2nd Edition. The MIT Press, 2001.
15. O. Shalev, N. Shavit: Split-Ordered Lists: Lock-Free Extensible Hash Tables. Journal of the ACM, vol. 53., no. 3., 2006.
16. William Pugh: Skip Lists: A Probabilistic Alternative to Balanced Trees. Communications ACM, volume 33, 1990.
17. William Pugh: Concurrent Maintenance of Skip Lists. 1990.
18. M. Herlihy, Y. Lev, V. Luchangco, N. Shavit: A Provably Correct Scalable Concurrent Skip List. OPODIS, 2006.
19. H. Kung, P. Lehman: Concurrent manipulation of binary search trees. ACM, 1980.
20. N. G. Bronson, J. Casper, H. Chafi, K. Olukotun: A Practical Concurrent Binary Search Tree. ACM, 2009.
21. F. Ellen, P. Fatourou, E. Ruppert, F. van Breugel: Non-blocking binary search trees. PODC, 2010.

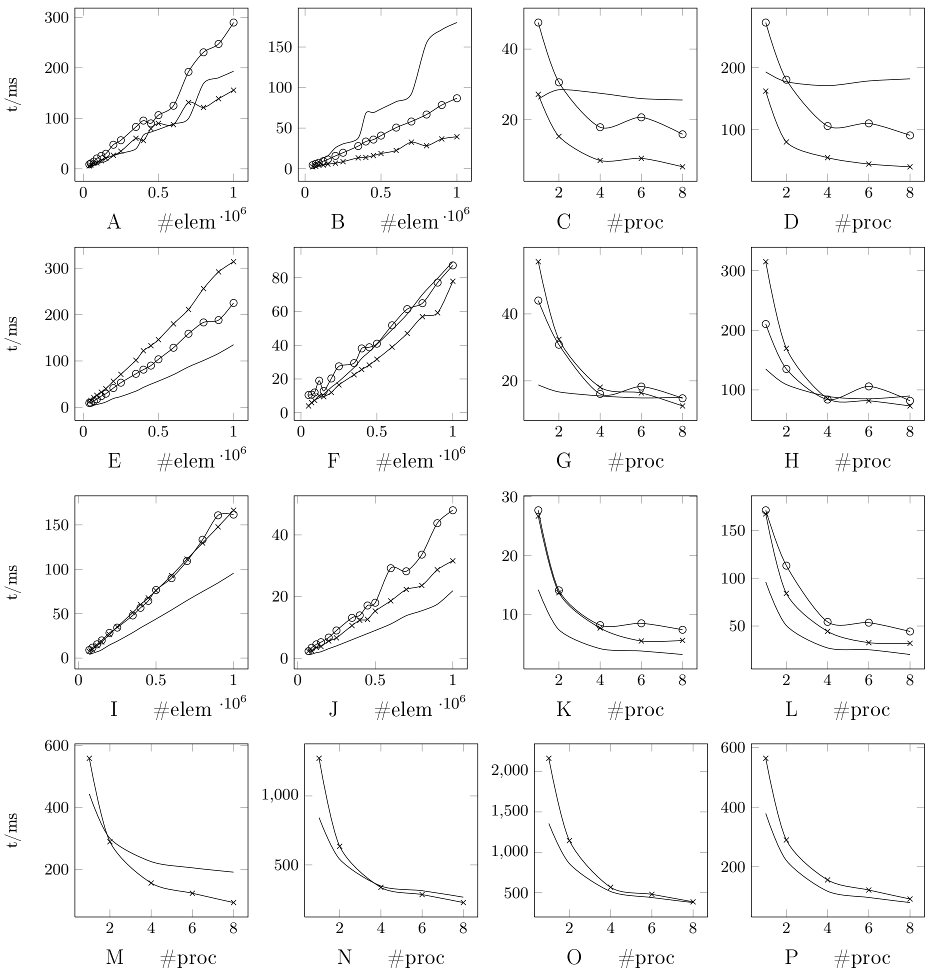

Fig. 6. Quad-core i7 microbenchmarks – *ConcurrentHashMap*(−), *ConcurrentSkipList*(○), *Ctrie*(×): A) *insert*, P=1; B) *insert*, P=8; C) *insert*, N=200k; D) *insert*, N=1M; E) *remove*, P=1; F) *remove*, P=8; G) *remove*, N=200k, H) *remove*, N=1M; I) *lookup*, P=1; J) *lookup*, P=8; K) *lookup*, N=200k; L) *lookup*, N=1M; M) *insert/lookup*, ratio=1/2, N=1M; N) *insert/lookup*, ratio=1/5, N=1M; O) *insert/lookup*, ratio=1/9, N=1M; P) *insert/lookup* with preallocated tables, ratio=1/2, N=1M