Cache-Aware Lock-Free Concurrent Hash Tries

Aleksandar Prokopec, Phil Bagwell, Martin Odersky

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
{firstname}.{lastname}@epfl.ch

Abstract

This report describes an implementation of a non-blocking concurrent shared-memory hash trie based on single-word compare-and-swap instructions. Insert, lookup and remove operations modifying different parts of the hash trie can be run independent of each other and do not contend. 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 which compare concurrent hash trie operations against the corresponding operations on other concurrent data structures, showing their performance and scalability.

1. Introduction

Many applications access data concurrently in the presence of multiple processors. Without proper synchronization concurrent access to data may result in errors in the user program. 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. A similar approach was taken in the dynamic array data structures [8]. In this paper we present and describe in detail a non-blocking implementation of the hash array mapped trie data structure.

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 and explain 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 which have 2 types of nodes – internal nodes and leaves. Leaves store key-value bindings. Internal nodes have a 2^k-way branching factor. In a straightforward implementation, each internal node is a 2^k-element array. Finding a key proceeds in the following manner. If the internal node is at the root, the initial k bits of the key hashcode are used as an index in the array. If the internal node is at the level l, then the bits k bits of the hashcode starting from the position $k * l$ 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^k. 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 k 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, k is 5 since that ensures that 32-bit integers can be used as bitmaps. An example hash trie is shown in Fig. 1A.

We want to preserve the nice properties of hash tries – space-efficiency, cache-awareness and the expected depth of $O(\log_{2^k}(n))$, where n is the number of elements stored in the trie and 2^k is the bitmap length. We also want to make hash tries a concurrent data structure which 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 this new data structure a *Ctrie*. In the remainder of this chapter we give examples of Ctrie operations.

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 which 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 which 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 which 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) which 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 which 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 which 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 which 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 which 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 which 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 which points to a tomb node a *tomb-inode*. We say that a tomb-inode 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-inode*.

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. We use logical symbols for boolean expressions. The pseudocode also contains pattern matching constructs which are used to match a node against its type. All occurences of pattern matching can be trivially replaced with a sequence of *if-then-else* statements – we use pattern matching for conciseness. The colon (:) in the pattern matching cases should be understood as *has type*. The keyword def denotes a procedure definition. Reads and compare-and-set instructions written in capitals are atomic – they occur at one point in time. This is a high level pseudocode and might not be optimal in all cases – the source code contains a more efficient implementation.

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

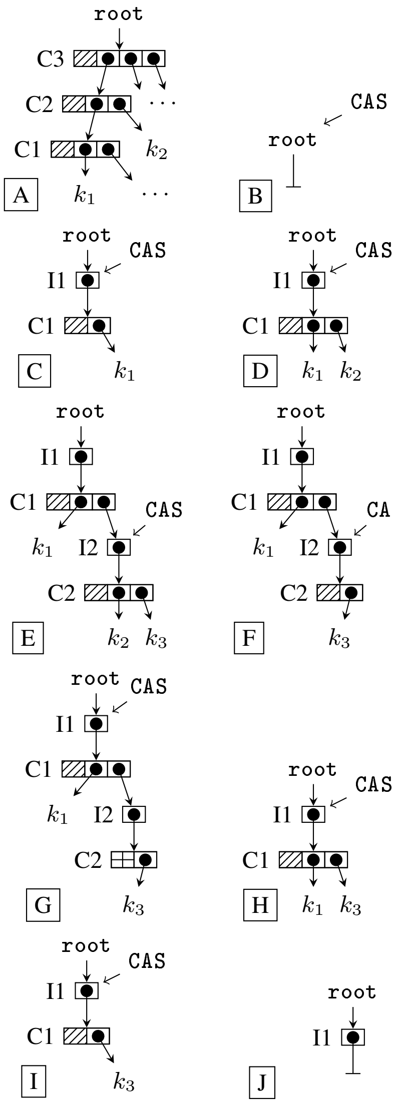

Figure 1. Hash trie and Ctrie examples

finds a key. If the `root` points to an `INode` which 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-inode then the node below the root inode is read (lines 35, 51 and 80), and we proceed casewise. If the node pointed at by the inode 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, 83). 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 inode 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 otherwise extend the trie below the `CNode`. A remove operation compares the keys – if they are the same it replaces the `CNode` with its updated version without the key.

```
          root: INode

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

Figure 2. Types and data structures

After a key was removed, the trie has to be contracted. A remove operation first attempts to create a tomb from the current `CNode`. It first reads the node below the current inode to check if it is still a `CNode`. It then calls `toWeakTombed` which 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-inodes is greater than 1, then it is the `CNode` itself – in this case 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-inode (alternatively, a *singleton*), the weak tomb is the tomb node with that binding. If the single branch below is another `CNode`, a weak tomb is a copy of the current `CNode` with the null-inodes removed.

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 133 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`, which checks if the inode is still reachable from its parent and then contracts the `CNode` below the parent - it removes the null-inode (line 149) or resurrects a tomb-inode into an `SNode` (line 153). 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 have originated from a remove operation which will attempt to contract the tomb node at some time in the future. Rather than waiting for that remove operation to do its work, the current operation should do the work of contracting the tomb itself, so it will invoke the `clean` operation on the parent inode. The `clean` operation will attempt to exchange the `CNode` below the parent inode 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-inodes (this is what the `filtered` call in the `toCompressed` procedure does) and with all the tomb-inodes 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 removal would have. After having fixed the Ctrie, the operation is repeated from the start.

```
def insert(k, v)
  r = READ(root)
  if r = null ∨ isNullInode(r) {
    scn = CNode(SNode(k, v, ⊥))
    nr = INode(scn)
    if !CAS(root, r, nr) insert(k, v)
  } else if ¬iinsert(r, k, v, 0, null)
    insert(k, v)

def remove(k)
  r = READ(root)
  if r = null return NOTFOUND
  else if isNullInode(r) {
    CAS(root, r, null)
    return remove(k)
  } else {
    res = iremove(r, k, 0, null)
    if res ≠ RESTART return res
    else remove(k)
  }

def lookup(k)
  r = READ(root)
  if r = null return NOTFOUND
  else if isNullInode(r) {
    CAS(root, r, null)
    return lookup(k)
  } else {
    res = ilookup(r, k, 0, null)
    if res ≠ RESTART return res
    else return lookup(k)
  }

def ilookup(i, k, lev, parent)
  READ(i.main) match {
  case cn: CNode =>
    flag, pos = flagpos(k.hc, lev, cn.bmp)
    if cn.bmp ⊙ flag = 0 return NOTFOUND
    cn.array(pos) match {
    case sin: INode =>
      return ilookup(sin, k, lev + W, i)
    case sn: SNode ∧ ¬sn.tomb =>
      if sn.k = k return sn.v
      else return NOTFOUND
    }
  case (sn: SNode ∧ sn.tomb) ∨ null =>
    if parent ≠ null clean(parent)
    return RESTART
  }
```

Figure 3. Basic operations I

```
def iinsert(i, k, v, lev, parent)
  READ(i.main) match {
  case cn: CNode =>
    flag, pos = flagpos(k.hc, lev, cn.bmp)
    if cn.bmp ⊙ flag = 0 {
      nsn = SNode(k, v, ⊥)
      narr = cn.array.inserted(pos, nsn)
      ncn = CNode(narr, bmp | flag)
      return CAS(i.main, cn, ncn)
    }
    cn.array(pos) match {
    case sin: INode =>
      return iinsert(sin, k, v, lev + W, i)
    case sn: SNode ∧ ¬sn.tomb =>
      nsn = SNode(k, v, ⊥)
      if sn.k = k {
        ncn = cn.updated(pos, nsn)
        return CAS(i.main, cn, ncn)
      } else {
        nin = INode(CNode(sn, nsn, lev + W))
        ncn = cn.updated(pos, nin)
        return CAS(i.main, cn, ncn)
      }
    }
  case (sn: SNode ∧ sn.tomb) ∨ null =>
    if parent ≠ null clean(parent)
    return ⊥
  }

def iremove(i, k, lev, parent)
  READ(i.main) match {
  case cn: CNode =>
    flag, pos = flagpos(k.hc, lev, cn.bmp)
    if cn.bmp ⊙ flag = 0 return NOTFOUND
    res = cn.array(pos) match {
    case sin: INode =>
      return iremove(sin, k, lev + W, i)
    case sn: SNode ∧ ¬sn.tomb =>
      if sn.k = k {
        narr = cn.array.removed(pos)
        ncn = CNode(narr, bmp ^ flag)
        if cn.array.length = 1 ncn = null
        if CAS(i.main, cn, ncn) return sn.v
        else return RESTART
      } else return NOTFOUND
    }
    if res = NOTFOUND ∨ res = RESTART return res
    if parent ne null ∧ tombCompress()
      contractParent(parent, in, k.hc, lev - W)
  case (sn: SNode ∧ sn.tomb) ∨ null =>
    if parent ≠ null clean(parent)
    return RESTART
  }
```

Figure 4. Basic operations II

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 – the complete proof is given in the appendix. 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 cnode $cn_{l,p}$ 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 inode $in_{l,p}$, $in_{l,p}.main$ is a cnode $cn_{l,p}$, a tombed snode $sn\dagger$ or $null$.

INV2 For every cnode 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 inode $in_{l+W,p\cdot r}$ or an snode.

INV4 If an entry in the array in $cn_{l,p}$ contains an snode sn, then p is the prefix of the hashcode $sn.k$.

INV5 If an inode $in_{l,p}$ contains an snode 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 snode 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

```
103 def toCompressed(cn)
104   num = bit#(cn.bmp)
105   if num = 1 ∧ isTombInode(cn.array(0))
106     return cn.array(0).main
107   ncn = cn.filtered(_.main ≠ null)
108   rarr = ncn.array.map(resurrect(_))
109   if bit#(ncn.bmp) > 0
110     return CNode(rarr, ncn.bmp)
111   else return null
112
113 def toWeakTombed(cn)
114   farr = cn.array.filtered(_.main ≠ null)
115   nbmp = cn.bmp.filtered(_.main ≠ null)
116   if farr.length > 1 return cn
117   if farr.length = 1
118     if isSingleton(farr(0))
119       return farr(0).tombed
120     else CNode(farr, nbmp)
121   return null
122
123 def clean(i)
124   m = READ(i.main)
125   if m ∈ CNode
126     CAS(i.main, m, toCompressed(m))
127
128 def tombCompress(i)
129   m = READ(i.main)
130   if m ∉ CNode return ⊥
131   mwt = toWeakTombed(m)
132   if m = mwt return ⊥
133   if CAS(i.main, m, mwt) mwt match {
134     case null ∨ (sn: SNode ∧ sn.tomb) =>
135       return ⊤
136     case _ => return ⊥
137   } else return tombCompress()
138
139 def contractParent(parent, i, hc, lev)
140   m, pm = READ(i.main), READ(parent.main)
141   pm match {
142   case cn: CNode =>
143     flag, pos = flagpos(k.hc, lev, cn.bmp)
144     if bmp ⊙ flag = 0 return
145     sub = cn.array(pos)
146     if sub ≠ i return
147     if m = null {
148       ncn = cn.removed(pos)
149       if !CAS(parent.main, cn, ncn)
150         contractParent(parent, i, hc, lev)
151     } else if isSingleton(m) {
152       ncn = cn.updated(pos, m.untombed)
153       if !CAS(parent.main, cn, ncn)
154         contractParent(parent, i, hc, lev)
155     }
156   case _ => return
157   }
```

Figure 5. Compression operations

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 inode in is either a null-inode or a tomb-inode at some time t_0 then $\forall t > t_0$ $in.main$ is not written. We refer to such inodes as nonlive.*

Lemma 2. *Cnodes and snodes 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 inode 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 cnode $cn_{i,p}$ 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 snode, cnode or $null$.*

Lemma 7. *Assume that a cnode 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 snode 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 92 succeeds at time t_1 after $in.main$ was read in line 80 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 which 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, 126, 133, 149 or 153 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, 124, 129, 140 or 140 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-inodes in this state, t the number of reachable tomb-inodes, l the number of live inodes, 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 which 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 126 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 133 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 149 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 153 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 which 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 – a new table is allocated and all the elements from the previous hash table have to be copied into the new hash table. More importantly, this implementation uses a global write lock during the resize phase – other threads adding new elements into the table have to wait until the resizal 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 resizal phase in concurrent hash tables, hence the ratio increases. This is shown in Fig. 7A-C. In the last benchmark (Fig. 7D) we preallocate the array for the concurrent hash table to avoid resizal 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] [22] [23]. 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 which 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]. 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 resizal is amortized against operations by one thread, this approach does not guarantee horizontal scalability. Lea developed an extensible hash algorithm which 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 which stores elements in a linked list. There are multiple levels of linked lists which allow efficient 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 which 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 which require a fixed number of locks to perform deletions [20]. Recently, the first nonblocking 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*, which reassigns a value from one key to another atomically. One research direction is supporting efficient aggregation operations on the keys and/or stored in the Ctrie. One such specific aggregation is the size of the Ctrie – an operation which might be useful indeed. The notion of having a size kept in one place in the Ctrie might, however, prove detrimental to the idea of distributing Ctrie operations throughout different parts of it in order to avoid contention.

Finally, we plan to develop an efficient lock-free snapshot operation for the concurrent trie which 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. A snapshot would allow traversing (in parallel) the elements of the Ctrie present at one point in time and modifying it during traversal in a way that the changes are visible only once the traversal ends. This might prove an efficient abstraction to express many iterative-style algorithms.

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.

[22] N. Shavit, A. Zemach: Scalable Concurrent Priority Queue Algorithms. IPDPS, 2000.

[23] M. Michael, M. Scott: Nonblocking Algorithms and Preemption-safe Locking on Multiprogrammed Shared-memory Multiprocessors. Journal of Parallel and Distributed Computing, 1998.

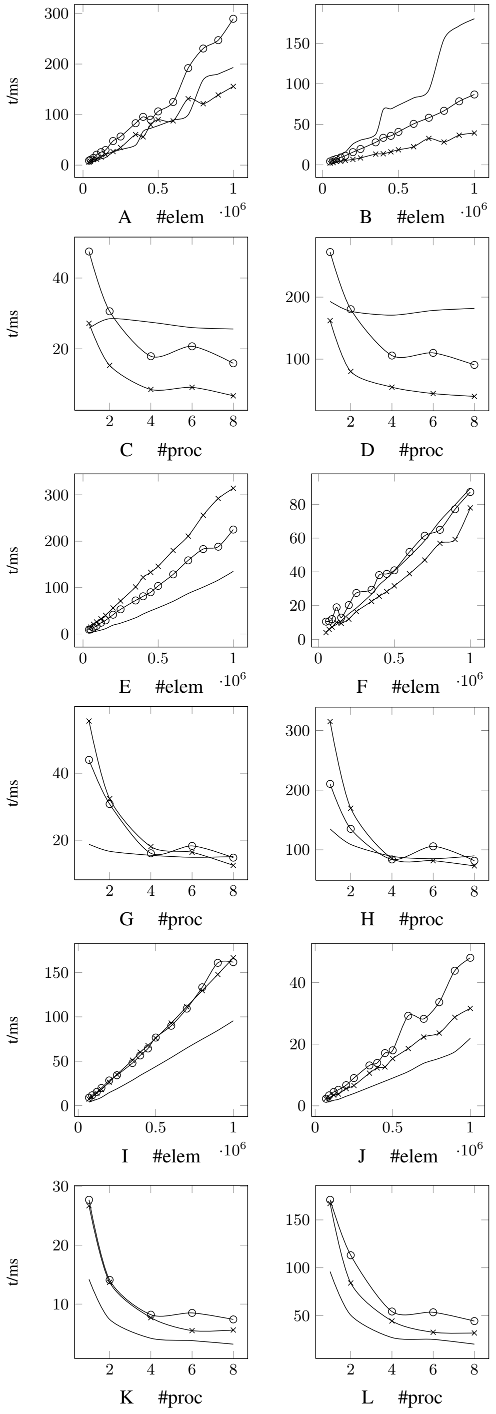

Figure 6. Quad-core i7 microbenchmarks – $ConcurrentHashMap(-)$, $ConcurrentSkipList(\circ)$, $Ctrie(\times)$: A) $insert$, P=1; B) $insert$, P=8; C) $insert$, N=200k; D) $insert$, N=1M; E) $remove$, P=1; F) $remove$, P=8; G) $remove$, N=200k, H) $remove$, N=1M; I) $lookup$, P=1; J) $lookup$, P=8; K) $lookup$, N=200k; L) $lookup$, N=1M

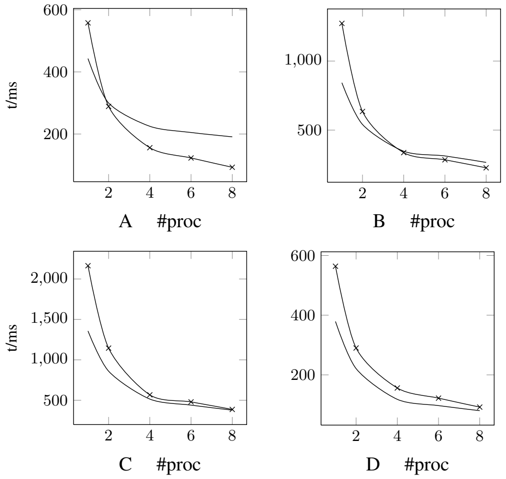

Figure 7. Quad-core i7 microbenchmarks – A) $insert/lookup$, ratio=1/2, N=1M; B) $insert/lookup$, ratio=1/5, N=1M; C) $insert/lookup$, ratio=1/9, N=1M; D) $insert/lookup$ with preallocated tables, ratio=1/2, N=1M

A. Proof of correctness

Definition 1 (Basics)**.** Value W is called the **branching width**. An **inode** in is a node holding a reference $main$ to other nodes. A **cnode** cn is a node holding a bitmap bmp and an set of references to other nodes called $array$. A cnode is k**-way** if $length(cn.array) = k$. An **snode** sn is a node holding a key k and a value v. An snode can be **tombed**, denoted by $sn\dagger$, meaning its tomb flag is set. A reference $cn.arr(r)$ in the $array$ defined as $array(\#(((1 << r) - 1) \odot cn.bmp))$, where $\#$ is the bitcount and $\odot$ is the bitwise-and operation. Any node $n_{l,p}$ is at **level** l if there are l/W cnodes on the simple path between itself and the root inode. **Hashcode chunk** of a key k at level l is defined as $m(l, k) = (hashcode(k) >> l) \mod 2^W$. A node at level 0 has a **hashcode prefix** $p = \epsilon$, where ϵ is an empty string. A node n at level $l + W$ has a hashcode prefix $p = q \cdot r$ if and only if it can be reached from the closest parent cnode $cn_{l,q}$ by following the reference $cn_{l,q}.arr(r)$. A reference $cn_{l,p}.sub(k)$ is defined as:

$$cn_{l,p}.sub(k) = \begin{cases} cn_{l,p}.arr(m(l,k)) & \text{if } cn_{l,p}.flg(m(l,k)) \\ null & \text{otherwise} \end{cases}$$

$$cn_{l,p}.flg(r) \Leftrightarrow cn_{l,p}.bmp \odot (1 \ll r) \neq 0$$

Definition 2 (Ctrie)**.** A **Ctrie** is defined as the reference $root$ to the root of the trie. A Ctrie **state** $\mathbb{S}$ is defined as the configuration of nodes reachable from the $root$ by following references in the nodes. A key is within the configuration if it is in a node reachable from the root. More formally, the relation $hasKey(in_{l,p}, k)$ on an inode in at the level l with a prefix p and a key k holds if and only if (several relations for readability):

$$\begin{aligned}
&holds(in_{l,p}, k) \Leftrightarrow in_{l,p}.main = sn : SNode \wedge sn.k = k \\
&holds(cn_{l,p}, k) \Leftrightarrow cn_{l,p}.sub(k) = sn : SNode \wedge sn.k = k \\
&hasKey(cn_{l,p}, k) \Leftrightarrow holds(cn_{l,p}, k) \vee \\
&(cn_{l,p}.sub(k) = in_{l+w,p \cdot m(l,k)} \wedge hasKey(in_{l+w,p \cdot m(l,k)}, k)) \\
&hasKey(in_{l,p}, k) \Leftrightarrow holds(in_{l,p}, k) \vee \\
&(in_{l,p}.main = cn_{l,p} : CNode \wedge hasKey(cn_{l,p}, k))
\end{aligned}$$

Definition 3. We define the following invariants for the Ctrie.

INV1 $inode_{l,p}.main = null | cnode_{l,p} | snode\dagger$
INV2 $\#(cn.bmp) = length(cn.array)$
INV3 $cn_{l,p}.flg(r) \neq 0 \Leftrightarrow cn_{l,p}.arr(r) \in \{sn, in_{l+W,p \cdot r}\}$
INV4 $cn_{l,p}.arr(r) = sn \Leftrightarrow hashcode(sn.k) = p \cdot r \cdot s$
INV5 $in_{l,p}.main = sn\dagger \Leftrightarrow hashcode(sn.k) = p \cdot r$

Definition 4 (Validity)**.** A state $\mathbb{S}$ is **valid** if and only if the invariants INV1-5 are true in the state $\mathbb{S}$.

Definition 5 (Abstract set)**.** An **abstract set** $\mathbb{A}$ is a mapping $K \Rightarrow \{\bot, \top\}$ which is true only for those keys which are a part of the abstract set, where K is the set of all keys. An **empty abstract set** $\varnothing$ is a mapping such that $\forall k, \varnothing(k) = \bot$. Abstract set operations are $insert(k, \mathbb{A}) = \mathbb{A}_1 : \forall k' \in \mathbb{A}_1, k' = k \vee k' \in \mathbb{A}$, $lookup(k, \mathbb{A}) = \top \Leftrightarrow k \in \mathbb{A}$ and $remove(k, \mathbb{A}) = \mathbb{A}_1 : k \notin \mathbb{A}_1 \wedge \forall k' \in \mathbb{A}, k \neq k' \Rightarrow k' \in \mathbb{A}$. Operations $insert$ and $remove$ are **destructive**.

Definition 6 (Consistency)**.** A Ctrie state $\mathbb{S}$ is **consistent** with an abstract set $\mathbb{A}$ if and only if $k \in \mathbb{A} \Leftrightarrow hasKey(Ctrie, k)$. A destructive Ctrie operation *op* is **consistent** with the corresponding abstract set operation *op'* if and only if applying *op* to a state $\mathbb{S}$ consistent with $\mathbb{A}$ changes the state into $\mathbb{S}'$ consistent with an abstract set $\mathbb{A}' = op(k, \mathbb{A})$. A Ctrie $lookup$ is **consistent** with the abstract set lookup if and only if it returns the same value as the abstract set $lookup$, given that the state $\mathbb{S}$ is consistent with $\mathbb{A}$. A **consistency change** is a change from state $\mathbb{S}$ to state $\mathbb{S}'$ of the Ctrie such that $\mathbb{S}$ is consistent with an abstract set $\mathbb{A}$ and $\mathbb{S}'$ is consistent with an abstract set $\mathbb{A}'$ and $\mathbb{A} \neq \mathbb{A}'$.

We point out that there are multiple valid states corresponding to the same abstract set.

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 $\mathbb{A}$.*

First, it is trivial to see that if the state $\mathbb{S}$ is valid, then the Ctrie is also consistent with some abstract set $\mathbb{A}$. Second, we prove the theorem using structural induction. As induction base, we take the empty Ctrie which is valid and consistent by definition. The induction hypothesis is that the Ctrie is valid and consistent at some time t. We use the hypothesis implicitly from this point on. Before proving the induction step, we introduce additional definitions and lemmas.

Definition 7. A node is **live** if and only if it is a cnode, a non-tombed snode or an inode whose $main$ reference points to a cnode. A **nonlive** node is a node which is not live. A **null-inode** is an inode with a $main$ set to $null$. A **tomb-inode** is an inode with a $main$ set to a tombed snode $sn\dagger$. A node is a **singleton** if it is an snode or an inode in such that $in.main = sn\dagger$, where $sn\dagger$ is tombed.

Lemma 1 (End of life)**.** *If an inode in is either a null-inode or a tomb-inode at some time t_0, then $\forall t > t_0$ $in.main$ is not written.*

Proof. For any inode in which becomes reachable in the Ctrie at some time t, all assignments to $in.main$ at any time $t_0 > t$ occur in a CAS instruction – we only have to inspect these writes.

Every CAS instruction on $in.main$ is preceeded by a check that the expected value of $in.main$ is a cnode. From the properties of CAS, it follows that if the current value is either $null$ or a tombed snode, the CAS will not succeed. Therefore, neither null-inodes nor tomb-inodes can be written to $in.main$. □

Lemma 2. *Cnodes and snodes are immutable – once created, they no longer change the value of their fields.*

Proof. Trivial inspection of the pseudocode reveals that k, v, $tomb$, bmp and $array$ are never assigned a value after an snode or a cnode was created. □

Definition 8. A **compression** ccn of a cnode cn seen at some time t_0 is a node such that:

- $ccn = sn\dagger$ if $length(cn.array) = 1$ and $cn.array(0).main = sn\dagger$ at t_0
- $ccn = null$ if $\forall i, cn.array(i).main = null$ at t_0 (including the case where $length(cn.array) = 0$)
- otherwise, ccn is a cnode obtained from cn so that at least those null-inodes existing at t_0 are removed and at least those tomb-inodes in existing at t_0 are resurrected - that is, replaced by untombed copies sn of $sn\dagger = in.main$

A **weak tombing** wtc of a cnode cn seen at some time t_0 is a node such that:

- $ccn = sn\dagger$ if $length(cn.array) = 1$ and $cn.array(0)$ is a tomb-inode or an snode at t_0
- $ccn = null$ if $\forall i, cn.array(i).main = null$ at t_0
- $ccn = cn$ if there is more than a single non-null-inode below at t_0
- otherwise, ccn is a one-way cnode obtained from cn such that all null-inodes existing at t_0 are removed

Lemma 3. *Methods $toCompressed$ and $toWeakTombed$ return the compression and weak tombing of a cnode cn, respectively.*

Proof. From lemma 2 we know that a cnode does not change values of bmp or $array$ once created. From lemma 1 we know that all the nodes that are nonlive at t_0 must be nonlive $\forall t > t_0$. Methods $toCompressed$ or $toWeakTombed$ scan the array of cn sequentially and make checks which are guaranteed to stay true if they pass – when these methods complete at some time $t > t_0$ they will have removed or resurrected at least those inodes that were nonlive at some point t_0 after the operation began. □

Lemma 4. *Invariants INV1, INV2 and INV3 are always preserved.*

Proof. INV1: Inode creation and every CAS instruction abide this invariant. There are no other writes to $main$.

INV2, INV3: Trivial inspection of the pseudocode shows that the creation of cnodes abides these invariants. From lemma 2 we know that cnodes are immutable. Therefore, these invariants are ensured during construction and do not change subsequently. □

Lemma 5. *If any CAS instruction makes an inode in unreachable from its parent at some time t, then in is nonlive at time t.*

Proof. We will show that all the inodes a CAS instruction could have made unreachable from their parents at some point t_1 were nonlive at some time $t_0 < t_1$. The proof then follows directly from lemma 1. We now analyze successful CAS instructions.

In lines 6, 14 and 26, if r is an inode and it is removed from the trie, then it must have been previously checked to be a null-inode in lines 3, 13 and 25, respectively.

In lines 58, 67 and 71, a cnode cn is replaced with a new cnode ncn which contains all the references to inodes as cn does, and possibly some more. These instructions do not make any inodes unreachable.

In line 92, a cnode cn is replaced with a new ncn which contains all the node references as cn but without one reference to an snode – all the inodes remain reachable.

In line 126, a cnode m is replaced with its compression mc – from lemma 3, mc may only be deprived of references to nonlive inodes.

In line 133, a cnode m is replaced with its weak tombing mwt – from lemma 3, mwt may only be deprived of references to nonlive inodes. □

Corollary 1. Lemma 5 has a consequence that any inode in can only be made unreachable in the Ctrie through modifications in their parent inode (or the root reference if in is referred by it). If there is a parent that refers to in, then that parent is live by definition. If the parent had been previously removed, lemma 5 tells us that the parent would have been nonlive at the time. From lemma 1 we know that the parent would remain nonlive afterwards. This is a contradiction.

Lemma 6. *If at some time t_1 an inode in is read by some thread (lines 2, 11, 23, 39, 60, 84, 145), followed by a read of cnode $cn = in.main$ in the same thread at time $t_2 > t_1$ (lines 35, 51, 80, 124, 129, 140), then in is reachable from the root at time t_2. Trivially, so is $in.main$.*

Proof. Assume, that inode in is not reachable from the root at t_2. That would mean that in was made unreachable at an earlier time $t_0 < t_2$. Corollary 1 says that in was then nonlive at t_0. However, from lemma 1 it follows that in must be nonlive for all times greater than t_0, including t_2. This is a contradiction – in is live at t_2, since it contains a cnode $cn = in.main$. □

Lemma 7 (Presence)**.** *Reading a cnode cn at some time t_0 and then $cn.sub(k)$ such that $k = sn.k$ at some time $t_1 > t_0$ means that the relation $hasKey(root, k)$ holds at time t_0. Trivially, k is then in the corresponding abstract set $\mathbb{A}$.*

Proof. We know from lemma 6 that the corresponding cnode cn was reachable at some time t_0. Lemma 2 tells us that cn and sn were the same $\forall t > t_0$. Therefore, sn was present in the array of cn at t_0, so it was reachable. Furthermore, $sn.k$ is the same $\forall t > t_0$. It follows that $hasKey(root, x)$ holds at time t_0. □

Definition 9. A **longest path** of nodes $\pi(h)$ for some hashcode h is the sequence of nodes from the root to a leaf of a valid Ctrie such that:

- if $root = null$ then $\pi(h) = \epsilon$
- if $root \neq null$ then the first node in $\pi(h)$ is $root$, which is an inode
- $\forall in \in \pi(h)$ if $in.main = cn$, then the next element in the path is cn
- $\forall in \in \pi(h)$ if $in.main = sn$, then the last element in the path is sn
- $\forall in \in \pi(h)$ if $in.main = null$, then the last element in the path is in
- $\forall cn_{l,p} \in \pi(h), h = p \cdot r \cdot s$ if $cn.flg(r) = \bot$, then the last element in the path is cn, otherwise the next element in the path is $cn.arr(r)$

Lemma 8 (Longest path)**.** *Assume that a non-empty Ctrie is in a valid state at some time t. The longest path of nodes $\pi(h)$ for some hashcode $h = r_0 \cdot r_1 \cdots r_n$ is a sequence $in_{0,\epsilon} \rightarrow cn_{0,\epsilon} \rightarrow in_{W,r_0} \rightarrow \ldots \rightarrow in_{W \cdot m, r_0 \cdots r_m} \rightarrow x$, where $x \in \{cn_{W \cdot m, r_0 \cdots r_m}, sn, cn_{W \cdot m, r_0 \cdots r_m} \rightarrow sn, null\}$.*

Proof. Trivially from the invariants and the definition of the longest path. □

Lemma 9 (Absence I)**.** *Assume that at some time t_0 $\exists cn = in.main$ for some node $in_{l,p}$ and the algorithm is searching for a key k. Reading a cnode cn at some time t_0 such that $cn.sub(k) = null$ and $hashcode(k) = p \cdot r \cdot s$ implies that the relation $hasKey(root, k)$ does not hold at time t_0. Trivially, k is not in the corresponding abstract set $\mathbb{A}$.*

Proof. Lemma 6 implies that in is in the configuration at time t_0, because $cn = cn_{l,p}$ such that $hashcode(k) = p \cdot r \cdot s$ is live. The induction hypothesis states that the Ctrie was valid at t_0. We prove that $hasKey(root, k)$ does not hold by contradiction. Assume there exists an snode sn such that $sn.k = k$. By lemma 8, sn can only be the last node of the longest path $\pi(h)$, and we know that cn is the last node in $\pi(h)$.

□

Lemma 10 (Absence II)**.** *Assume that the algorithm is searching for a key k. Reading a live snode sn at some time t_0 and then $x = sn.k \neq k$ at some time $t_1 > t_0$ means that the relation $hasKey(root, x)$ does not hold at time t_0. Trivially, k is not in the corresponding abstract set $\mathbb{A}$.*

Proof. Contradiction similar to the one in the previous lemma. □

Lemma 11 (Absence III)**.** *Assume that the $root$ reference is read in r at t_0 and r is positively compared to $null$ at $t_1 > t_0$. Then $\forall k, hasKey(root, k)$ does not hold at t_0. Trivially, the Ctrie is consistent with the empty abstract set $\varnothing$.*

Proof. Local variable r has the same value $\forall t \geq t_0$. Therefore, at t_0 $root = null$. The rest is trivial. □

Lemma 12 (Fastening)**.** *1. Assume that one of the CAS instructions in lines 6, 14 and 26 succeeds at time t_1 after r was determined to be a nonlive inode in one of the lines 3, 13 or 25, respectively, at time t_0. Then $\forall t, t_0 \leq t < t_1$, relation $hasKey(root, k)$ does not hold for any key. If r is null, then $\exists \delta > 0 \forall t, t_1 - \delta < t < t_1$ $hasKey$ does not hold for any key.*

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

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

4. Assume that the CAS instruction in line 92 succeeds at time t_1 after $in.main$ was read in line 80 at time t_0. The $\forall t, t_0 \leq t < t_1$, relation $hasKey(root, k)$ holds.

Proof. The algorithm never creates a reference to a newly allocated memory areas unless that memory area has been previously reclaimed. Although it is possible to extend the pseudocode with memory management directives, we omit memory-reclamation from the pseudocode and assume the presence of a garbage collector which does not reclaim memory areas as long as there are references to them reachable from the program. In the pseudocode, CAS instructions always work on memory locations holding references – $CAS(x, r, r')$ takes a reference r to a memory area allocated for nodes as its expected value, meaning that a reference r that is reachable in the program exists from the time t_0 when it was read until $CAS(x, r, r')$ was invoked at t_1. On the other hand, the new value for the CAS is in all cases a newly allocated object. In the presence of a garbage collector with the specified properties, a new object cannot be allocated in any of the areas still being referred to. It follows that if a CAS succeeds at time t_1, then $\forall t, t_0 \leq t < t_1$, where t_0 is the time of reading a reference and t_1 is the time when CAS occurs, the corresponding memory location x had the same value r.

We now analyze specific cases from the lemma statement:

1. We know that $\forall t, t_0 \leq t < t_1$ the root reference has a reference r to the same inode in. We assumed that r is nonlive at t_0. From lemma 1 it follows that r remains nonlive until time t_1. By the definition of $hasKey$, the relation does not hold for any key from $\forall t, t_0 \leq t < t_1$. Case where r is $null$ is proved similarly.

2. From lemma 8 we know that for some hashcode $h = hashcode(k)$ there exists a longest path of nodes $\pi(h) = in_{0,\epsilon} \rightarrow \ldots \rightarrow cn_{l,p}$ such that $h = p \cdot r \cdot s$ and that sn such that $sn.k = k$ cannot be a part of this path – it could only be referenced by $cn_{l,p}.sub(k)$ of the last cnode in the path. We know that $\forall t, t_0 \leq t < t_1$ reference cn points to the same cnode. We know from 2 that cnodes are immutable. The check to $cn.bmp$ preceeding the CAS ensures that $\forall t, t_0 \leq t < t_1$ $cn.sub(k) = null$. In the other case, we check that the key k is not contained in sn. We know from 5 that cn is reachable during this time, because in is reachable. Therefore, $hasKey(root, k)$ does not hold $\forall t, t_0 \leq t < t_1$.

3., 4. We know that $\forall t, t_0 \leq t < t_1$ reference cn points to the same cnode. Cnode cn is reachable as long as its parent inode in is reachable. We know that in is reachable by lemma 5, since in is live $\forall t, t_0 \leq t < t_1$. We know that cn is immutable by lemma 2 and that it contains a reference to sn such that $sn.k = k$. Therefore, sn is reachable and $hasKey(root, k)$ holds $\forall t, t_0 \leq t < t_1$. □

Lemma 13. *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 12 induce a change into a valid state which is consistent with the abstract set semantics.*

Proof. From lemma 12, we know that a successful CAS in line 6 means that the Ctrie was consistent with an empty abstract set $\varnothing$ up to some time t_1. After that time, the Ctrie is consistent with the abstract set $\mathbb{A} = k$. Successful CAS instructions in lines 14 and 26 mean that the Ctrie was consistent with an empty abstract set $\varnothing$ up to some time t_1 and are also consistent with $\varnothing$ at t_1.

Observe a successful CAS in line 58 at some time t_1 after cn was read in line 51 at time $t_0 < t_1$. From lemma 12 we know that $\forall t, t_0 \leq t < t_1$, relation $hasKey(root, k)$ does not hold. If the last CAS instruction in the Ctrie occuring before the CAS in line 126 was at $t_\delta = t_1 - \delta$, then we know that $\forall t, \max(t_0, t_\delta) \leq t < t_1$ the $hasKey$ relation does not change. We know that at t_1 cn is replaced with a new cnode with a reference to a new snode sn such that $sn.k = k$, so at t_1 relation $hasKey(root, k)$ holds. Consequently, up to $\forall t, \max(t_0, t_\delta) \leq t < t_1$ the Ctrie is consistent with an abstract set $\mathbb{A}$ and at t_1 it is consistent with an abstract set $\mathbb{A} \cup \{k\}$. Validity is trivial.

Proofs for the CAS instructions in lines 67, 71 and 92 are similar. □

Lemma 14. *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.*

Proof. Operations $clean$, $tombCompress$ and $contractParent$ are atomic - their linearization point is the first successful CAS instruction occuring at t_1. We know from lemma 3 that methods $toCompressed$ and $toWeakTombed$ produce a compression and a weak tombing of a cnode, respectively.

We first prove the property $\exists k, hasKey(cn, k) \Rightarrow hasKey(f(cn), k)$, where f is either a compression or a weak tombing. We know from their respective definitions that the resulting cnode $ncn = f(cn)$ or the result $null = f(cn)$ may only omit nonlive inodes from cn. Omitting a null-inode omits no key. Omitting a tomb-inode may omit exactly one key, but that is compensated by adding new snodes – $sn\dagger$ in the case of a one-way node or, with compression, resurrected copies sn of removed inodes in such that $in.main = sn\dagger$. Therefore, the $hasKey$ relation is exactly the same for both cn and $f(cn)$.

We only have to look at cases where CAS instructions succeed. If CAS in line 126 at time t_1 succeeds, then $\forall t, t_0 < t < t_1$ $in.main = cn$ and at t_1 $in.main = toCompressed(cn)$. Assume there is some time $t_\delta = t_1 - \delta$ at which the last CAS instruction in the Ctrie occuring before the CAS in line 126 occurs. Then $\forall t, \max(t_0, t_\delta) \leq t < t_1$ the $hasKey$ relation does not change. Additionally, it does not change at t_1, as shown above. Therefore, the Ctrie remains consistent with the abstract set $\mathbb{A}$. Validity is trivial.

Proof for $tombCompress$ and $contractParent$ is similar. □

Corollary 2. From lemmas 13 and 14 it follows that invariants INV4 and INV5 are always preserved.

Safety. We proved at this point that the algorithm is safe - Ctrie is always in a valid (lemma 4 and corollary 2) state consistent with some abstract set. All operations are consistent with the abstract set semantics (lemmas 7, 9, 10, 11 13 and 14). □

Theorem 2 (Linearizability)**.** *Operations $insert$, $lookup$ and $remove$ are linearizable.*

Linearizability. An operation is linearizable if we can identify its linearization point. The linearization point is a single point in time when the consistency of the Ctrie changes. The CAS instruction itself is linearizable, as well as atomic reads. It is known that a single invocation of a linearizable instruction has a linearization point.

1. We know from lemma 14 that operation $clean$ does not change the state of the corresponding abstract set. Operation $clean$ is followed by a restart of the operation it was called from and is not preceeded by a consistency change – all successful writes in the $insert$ and $iinsert$ that change the consistency of the Ctrie result in termination.

CAS in line 6 that succeeds at t_1 immediately returns. By lemma 13, $\exists \delta > 0 \forall t, t_1 - \delta < t < t_1$ the Ctrie is consistent with an empty abstract set $\varnothing$, and at t_1 it is consistent with $\mathbb{A} = \{k\}$. If this is the first invocation of $insert$, then the CAS is the first and the last write with consistent semantics. If $insert$ has been recursively called, then it has not been preceeded by a consistency change – no successful CAS instruction in $iinsert$ is followed by a recursive call to the method $insert$. Therefore, it is the linearization point.

CAS in line 58 that succeeds at t_1 immediately returns. By lemma 13, $\exists \delta > 0 \forall t, t_1 - \delta < t < t_1$ the Ctrie is consistent with an empty abstract set $\mathbb{A}$ and at t_1 it is consistent with $\mathbb{A} \cup \{k\}$. If this is the first invocation of $iinsert$, then the CAS is the first and the last write with consistent semantics. If $iinsert$ has been recursively called, then it was preceeded by an $insert$ or $iinsert$. We have shown that if its preceeded by a call to $insert$, then there have been no preceeding consistency changes. If it was preceeded by $iinsert$, then there has been no write in the previous $iinsert$ invocation. Therefore, it is the linearization point.

Similar arguments hold for CAS instructions in lines 67 and 71. It follows that if some CAS instruction in the $insert$ invocation is successful, then it is the only successful CAS instruction. Therefore, $insert$ is linearizable.

2. Operation $clean$ is not preceeded by a write that results in a consistency change and does not change the consistency of the Ctrie.

Assume that a check in line 25 succeeds. The state of the local variable r does not change $\forall t > t_0$ where t_0 is the atomic read in the preceeding line 23. The linearization point is then the read at t_0, by lemma 11.

Assume that a CAS in line 26 succeeds at t_1. By lemma 13, $\exists \delta > 0 \forall t, t_1 - \delta < t < t_1$ the Ctrie is consistent with an empty abstract set $\varnothing$, and at t_1 it is consistent with $\varnothing$. Therefore, this write does not result in consistency change and is not preceeded by consistency changes. This write is followed by the restart of the operation.

Assume that a node m is read in line 35 at t_0. By lemma 2, if $cn.sub(k) = null$ at t_1 then $\forall t, cn.sub(k) = null$. By corollary 1, cn is reachable at t_0, so at t_0 the relation $hasKey(root, k)$ does not hold. The read at t_0 is not preceeded by a consistency changing write and followed by a termination of the $lookup$ so it is a linearization point if the method returns in line 38. By similar reasoning, if the operation returns in lines 43 or 44, the read in line 35 is the linearization point..

We have identified linearization points for the $lookup$, therefore $lookup$ is linearizable.

3. Operation $clean$ is not preceeded by a write that results in a consistency change and does not change the consistency of the Ctrie.

By lemma 14 operations $tombCompress$ and $contractParent$ do not cause a consistency change. Furthermore, they are only followed by calls to $tombCompress$ and $contractParent$ and the termination of the operation.

Assume that the check in line 13 succeeds after the read in line 11 at time t_0. By applying the same reasoning as for $lookup$ above, the read at time t_0 is the linearization point.

Assume CAS in line 14 succeeds at t_1. We apply the same reasoning as for $lookup$ above – this instruction does not change the consistency of the Ctrie and is followed by a restart of the operation.

Assume that a node m is read in line 80 at t_0. By similar reasoning as with $lookup$ above, the read in line 80 is a linearization point if the method returns in either of the lines 83 or 94.

Assume that the CAS in line 92 succeeds at time t_0. By lemma 13, $\exists \delta > 0 \forall t, t_1 - \delta < t < t_1$ the Ctrie is consistent with an empty abstract set $\mathbb{A}$ and at t_1 it is consistent with $\mathbb{A} \setminus \{k\}$. This write is not preceeded by consistency changing writes and followed only by $tombCompress$ and $contractParent$ which also do not change consistency. Therefore, it is a linearization point.

We have identified linearization points for the $remove$, therefore $remove$ is linearizable. □

Definition 10. Assume that a multiple number of threads are invoking a concurrent operation *op*. The concurrent operation *op* is **lock-free** if and only if after a finite number of thread execution steps some thread completes the operation.

Theorem 3 (Lock-freedom)**.** *Ctrie operations insert, lookup and remove are lock-free.*

The rough idea of the proof is the following. To prove lock-freedom we will first show that there is a finite number of steps between state changes. Then we define a space of possible states and show that there can only be finitely many successful CAS instructions which do not result in a consistency change. We have shown in lemmas 13 and 14 that only CAS instructions in lines 14, 26, 133, 149 and 153 do not cause a consistency change. We proceed by introducing additional definitions and prooving the necessary lemmas. In all cases, we assume there has been no state change which is a consistency change, otherwise that would mean that some operation was completed.

Lemma 15. *The root is never a tomb-inode.*

Proof. A tomb-inode can only be assigned to $in.main$ of some in in $clean$ and $tombCompress$. Neither $clean$ nor $tombCompress$ are ever called for the in in the root of the Ctrie, as they are preceeded by the check if $parent$ is different than $null$. □

Lemma 16. *If a CAS that does not cause a consistency change in one of the lines 58, 67, 71, 126, 133, 149 or 153 fails at some time t_1, then there has been a state change since the time t_0 when a respective read in one of the lines 51, 51, 51, 124, 129, 140 or 140 occured. Trivially, the state change preceeded the CAS by a finite number of execution steps.*

Proof. The configuration of nodes reachable from the root has changed, since the corresponding $in.main$ has changed. Therefore, the state has changed by definition. □

Lemma 17. *In each operation there is a finite number of execution steps between consecutive CAS instructions.*

Proof. The $ilookup$ and $iinsert$ operations have a finite number of executions steps. There are no loops in the pseudocode for $ilookup$ in $iinsert$, the recursive calls to them occur on the lower level of the trie and the trie depth is bound – no non-consistency changing CAS increases the depth of the trie.

The $lookup$ operation is restarted if and only if there has been a CAS in line 26 or if $clean$ (which contains a CAS) is called in $ilookup$. If $clean$ was not called in $ilookup$ after the check that the parent is not $null$ at t_0, then $root$ was in such that $in.main = null$ at t_0 (it is not tombed by lemma 15). Assuming there has been no state change, the CAS will occur in the next recursive call to $lookup$.

The $insert$ operation is restarted if and only if there has been a CAS in line 6 or if $clean$ (which contains a CAS) is called in $iinsert$. If $clean$ was not called in $iinsert$ after the check that the parent is not $null$ at t_0, then $root$ was in such that $in.main = null$ at t_0. Assuming no state change, a CAS will occur in the next recursive call to $insert$.

The $insert$ operation can also be restarted due to a preceeding failed CAS in lines 58, 67 or 71. By lemma 16, there must have been a state change in this case.

In $iremove$, calls to $tombCompress$ and $contractParent$ contain no loops, but are recursive. In case they restart themselves, a CAS is invoked at least once. Between these CAS instructions there is a finite number of execution steps.

A similar analysis as for $lookup$ above can be applied to the first phase of $remove$ which consists of all the execution steps preceeding a successful CAS in line 92. The number of times $tombCompress$ and $contractParent$ from the $iremove$ in the cleanup phase is bound by the depth of the trie and there is a finite number of execution steps between them. Once the root is reached, $remove$ completes.

Therefore, all operations have a finite number of executions steps between consecutive CAS instructions, assuming that the state has not changed since the last CAS instruction. □

Corollary 3. The consequence of lemmas 17 and 16 is that there is a finite number of execution steps between two state changes. At any point during the execution of the operation we know that the next CAS instruction is due in a finite number of execution steps (lemma 17). From lemmas 13 and 14 we know that if a CAS succeeds, it changes the state. From lemma 16 we know that if the CAS fails, the state was changed by someone else.

We remark at this point that corollary 3 does not imply that there is a finite number of execution steps between two operations. A state change is not necessarily a consistency change.

Definition 11. Let there at some time t_0 be a 1-way cnode cn such that $cn.array(0) = in$ and $in.main = sn\dagger$ where $sn\dagger$ is tombed or, alternatively, cn is a 0-way node. We call such cn a **single tip of length** 1. Let there at some time t_0 be a 1-way cnode cn such that $cn.array(0) = cn'$ and cn' is a single tip of length k. We call such cn a **single tip of length** $k + 1$.

Definition 12. The **total path length** d is the sum of the lengths of all the paths from the root to some leaf.

Definition 13. Assume the Ctrie is in a valid state. Let n be the number of reachable null-inodes in this state, t the number of reachable tomb-inodes, l the number of live inodes, 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 18. *Observe all CAS instructions which 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 126 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 133 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 149 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 153 changes the state from $\mathbb{S}_{n,t,l,r}$ to $\mathbb{S}_{n,t-1,l,r+j,d-1}$ where $j \geq 0$.

Proof. We have shown in lemma 14 that the CAS in line 126 does not change the number of live nodes. In lemma 3 we have shown that $toCompressed$ returns a compression of the cnode cn which replaces cn at $in.main$ at time t.

Provided there is at least one single tip immediately before time t, the compression of the cnode cn can omit at most one single tip, decreasing r by one. Omitting a single tip will also decrease d by one. If it is removing a single tip which is 1-way cnode, it will create a new null-inode in the trie, hence the $n + j$.

Provided there are at least k null-inodes and j tomb-inodes in the trie, compression may omit up to k null-inodes and up to j tomb-inodes. Value d may decrease in the new state. If both k and j are 0, then the state must have changed since a nonlive inode was detected prior to calling $clean$.

This proves the statement for CAS in line 126, the rest are either trivial or can be proved by applying a similar reasoning. □

Lemma 19. *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.*

Proof. Assume that there are $m \leq n$ threads in the $clean$ operation or in the cleanup phase of the $remove$. Since the state is clean, there are no nonlive inodes, so it is trivial to show that none of these m threads will invoke a CAS after their next CAS (which will be unsuccessful). This means that these m threads will either complete in a finite number of steps or restart the original operation after a finite number of steps. From this point on, as shown in lemma 17, the first CAS will be executed after a finite number of steps. Since the state is clean, there are no more nonlive inodes, so $clean$ will not be invoked. Therefore, the first CAS will result in a consistency change. Since it is the first CAS, it will also be successful. □

Lock-freedom. Assume we start in some state $\mathbb{S}_{n,t,l,r,d}$. We prove there are a finite number of state changes before reaching a clean state by contradiction. Assume there is an infinite sequence of state changes. We now use results from lemma 18. In this infinite sequence, a state change which decreases d may occur only finitely many times, since no state change increases d. After this finitely many state changes $d = 0$ so the sequence can contain no more state changes which decrease d. We apply the same reasoning to r – no available state change can increase the value of r, so after finitely many steps $r = 0$. At this point, we can only apply state changes which decrease n. After finitely many state changes $n = 0$. Therefore, the assumption is wrong – such an infinite sequence of state changes does not exist.

From corollary 3 there are a finite number of execution steps between state changes, so there are a finite number of execution steps before reaching a clean state. By lemma 19, if the Ctrie is in a clean state, then there are an additional finite number of steps until a consistency change occurs.

This proves that some operation completes after a finite number of steps, so all Ctrie operations are lock-free. □

Definition 14. A **tip** is a cnode cn such that it contains at most one reference to a tomb-inode or an snode. It may contain zero or more null-inodes, but no cnodes. If the first ancestor cnode is k-way where $k > 1$, then the tip has length 1.

The compression operations are designed so that they collect as many null-inodes as possible, and to prevent that there are tips. We now prove that they ensure that there are no tips in the trie.

Theorem 4 (Compactness). *Assume all $remove$ operations have completed execution. Then there is at most* 1 *tip of length 1 in the trie.*

Compactness. Assume that at some inode in in the trie some $remove$ operation created a tip cn at time t_0 by invoking a CAS instruction in line 92. The $remove$ operation then repeatedly tries to replace the cn with a new node mwt such that mwt is a weak tombing of cn. It stops in 2 cases.

If at some time $t_1 > t_0$ the operation detects that in is not a tip, it will stop. If in is not a tip, then it can safely abort the compression operation, since only some other $remove$ operation performing a successful CAS in line 92 at some time $t_2 > t_1$ can create a tip, and that $remove$ operation will invoke the compression again.

If at some time $t_1 > t_0$ the CAS in line 133 succeeds, then in will become nonlive – no longer a tip. Therefore, by lemma 1 in does not change the value of $in.main$ $\forall t > t_0$, and all modifications to the values in that branch must occur at the first inode ancestor of in – its $parent$. Method $contractParent$ is called next in this case. If it finds that $bmp \odot flag = 0$ or $sub \neq in$, then in is no longer reachable, so there are no more tips created by the current $remove$ operation at in – some other $remove$ operation may create a tip after t_1 at the same level and prefix as in, but in this case subsequent operations will be responsible for removing that tip. If in is reachable, a null-inode is removed from the cnode below the $parent$ (line 149) or a tomb-inode is resurrected into an snode (line 153). Notice that this can create a tip one level higher, but the whole procedure is repeated one level above for this reason.

The only case where we do not invoke $tombCompress$ is the root, where $parent = null$. The root can, therefore, contain at most 1 tip of length 1. □