Concurrent Tries with Efficient Non-Blocking Snapshots

Aleksandar Prokopec
EPFL
aleksandar.prokopec@epfl.ch

Nathan G. Bronson
Stanford
ngbronson@gmail.com

Phil Bagwell
Typesafe
phil.bagwell@typesafe.com

Martin Odersky
EPFL
martin.odersky@epfl.ch

Abstract

We describe a non-blocking concurrent hash trie based on shared-memory single-word compare-and-swap instructions. The hash trie supports standard mutable lock-free operations such as insertion, removal, lookup and their conditional variants. To ensure space-efficiency, removal operations compress the trie when necessary.

We show how to implement an efficient lock-free snapshot operation for concurrent hash tries. The snapshot operation uses a single-word compare-and-swap and avoids copying the data structure eagerly. Snapshots are used to implement consistent iterators and a linearizable size retrieval. We compare concurrent hash trie performance with other concurrent data structures and evaluate the performance of the snapshot operation.

Categories and Subject Descriptors E.1 [*Data structures*]: Trees

General Terms Algorithms

Keywords hash trie, concurrent data structure, snapshot, non-blocking

1. Introduction

When designing concurrent data structures, lock-freedom is an important concern. Lock-free data structures generally guarantee better robustness than their lock-based variants [11], as they are unaffected by thread delays. A fundamental difference is that lock-free data structures can continue to work correctly in the presence of failures, whereas a failure may prevent progress indefinitely with a lock-based data structure.

While obtaining a consistent view of the contents of the data structure at a single point in time is a trivial matter for data structures accessed sequentially, it is not clear how to do this in a non-blocking concurrent setting. A consistent view of the data structure at a single point in time is called a *snapshot*. Snapshots can be used to implement operations requiring global information about the data structure – in this case, their performance is limited by the performance of the snasphot.

Our contributions are the following:

1. We describe a complete lock-free concurrent hash trie data structure implementation for a shared-memory system based on single-word compare-and-swap instructions.

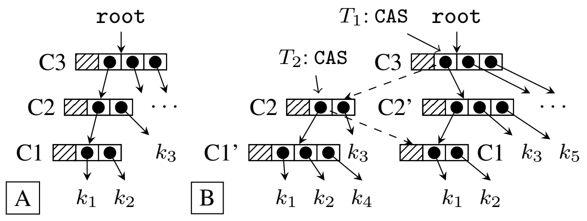

Figure 1. Hash tries

2. We introduce a non-blocking, atomic constant-time snapshot operation. We show how to use them to implement atomic size retrieval, consistent iterators and an atomic clear operation.
3. We present benchmarks that compare performance of concurrent tries against other concurrent data structures across different architectures.

Section 2 illustrates usefulness of snapshots. Section 3 describes basic operations on concurrent tries. Section 4 describes snapshots. Section 5 presents various benchmarks. Section 6 contains related work and Section 7 concludes.

2. Motivation

Most stock concurrent collection implementations include operations such as the atomic lookup, insert and remove operations. Operations that require global data structure information or induce a global change in the data structure, such as size retrieval, iterator creation or deleting all the elements are typically implemented with no atomicity guarantees (e.g. the Java `ConcurrentHashMap` and the `ConcurrentSkipListMap` [14]) or require a global lock. Ideally, these operations should have a constant running time and be nonblocking. Given an atomic snapshot, implementing these operations seems to be trivial.

Collection frameworks such as Intel TBB or STAPL parallelize bulk operations on data. They do this by relying on iterators that traverse disjunct parts of the collection. Many algorithms exhibit an interesting interplay of parallel traversal and concurrent updates. One such example is the PageRank algorithm, implemented using Scala parallel collections [21] in Figure 2. In a nutshell, this iterative algorithm updates the rank of the pages until the rank converges. The rank is updated based on the last known rank of the pages linking to the current page (line 4). Once the rank becomes smaller than some predefined constant, the page is removed from the set of pages being processed (line 5). The *for* loop that does the updates is executed in parallel. After the loop completes, the arrays containing the previous and the next rank are swapped in line 7, and the next iteration commences if there are pages left.

The main point about this algorithm is that the set of pages being iterated is updated by the remove operation during the parallel traversal. This is where most concurrent data structures prove inadequate for implementing this kind of algorithms – an iterator may or may not reflect the concurrent updates. Scala parallel collections can remedy this by removing the test in line 5 and adding another parallel operation `filter` to create a new set of pages without those that converged – this new set is traversed in the next iteration. The downside of this is that if only a few pages converge during an iteration then almost the entire set needs to be copied. If the iterators used for parallel traversal reflected only the elements present when the operation began, there would be no need for this. We show in Section 5 that avoiding the additional `filter` phase enhances performance.

```
1  while (pages.nonEmpty) {
2    for (page <- pages.par) {
3      val sum = page.incoming.sumBy(p => last(p) / p.links)
4      next(page) = (1 - damp) / N + damp * sum
5      if (next(page) - last(page) < eps) pages.remove(page)
6    }
7    swap(next, last)
8  }
```

Figure 2. Parallel PageRank implementation

3. Basic operations

Hash array mapped tries (or simply hash tries) described previously by Bagwell [2] are trees composed of 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 as follows. If the internal node is at the level l, then the W bits of the hashcode starting from the position $W * l$ are used as an index to the appropriate branch in the array. This is repeated until a leaf or an empty entry is found. Insertion uses the key to find an empty entry or a leaf. It creates a new leaf with the key if an empty entry is found. Otherwise, the key in the leaf is compared against the key being inserted. If they are equal, the existing leaf is replaced with a new one. If they are not equal (meaning their hashcode prefixes are the same) then the hash trie is extended with a new level.

Bagwell describes an implementation that is more space-efficient [2]. Each internal node contains a bitmap of length 2^W. If a bit is set, then the corresponding array entry contains either a branch or a leaf. The array length is equal to the number of bits in the bitmap. The corresponding array index for a bit on position i in the bitmap bmp is calculated as $\#((i-1) \odot bmp)$, where $\#(\cdot)$ is the bitcount and $\odot$ is a bitwise AND operation. The W bits of the hashcode relevant at some level l are used to compute the bit position 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. Figure 1A shows a hash trie example.

The goal is to create a concurrent data structure that preserves the space-efficiency of hash tries and the expected depth of $O(log_{2^W}(n))$. Lookup, insert and remove will be based solely on CAS instructions and have the lock-freedom property. Remove operations must ensure that the trie is kept as compact as possible. Finally, to support linearizable lock-free iteration and size retrievals, the data structure must support an efficient snapshot operation. We will call this data structure a *Ctrie*.

Intuitively, a concurrent insertion operation could start by locating the internal node it needs to modify and then create a copy of that node with both the bitmap and the array updated with a reference to the key being inserted. A reference to the newly created node could then be written into the array of the parent node using the CAS instruction. Unfortunately, this approach does not work. The fundamental problem here is due to races between an insertion of a key into some node $C1$ and an insertion of another key into its parent node $C2$. One scenario where such a race happens is shown in Figure 1. Assume we have a hash trie from the Figure 1A and that a thread T_1 decides to insert a key k_5 into the node $C2$ and creates an updated version of $C2$ called $C2'$. It must then do a CAS on the first entry in the internal node $C3$ with the expected value $C2$ and the new value $C2'$. Assume that another thread T_2 decides to insert a key k_4 into the node $C1$ before this CAS. It will create an updated version of $C1$ called $C1'$ and then do a CAS on the first entry of the array of $C2$ – the updated node $C1'$ will not be reachable from the updated node $C2'$. After both threads complete their CAS operations, the trie will correspond to the one shown in Figure 1B, where the dashed arrows represent the state of the branches before the CASes. The key k_4 inserted by the thread T_2 is lost.

We solve this problem by introducing indirection nodes, or I-nodes, which remain present in the Ctrie even as nodes above and below change. The CAS instruction is performed on the I-node instead of on the internal node array. We show that this eliminates the race between insertions on different levels.

The second fundamental problem has to do with the remove operations. Insert operations extend the Ctrie with additional levels. A sequence of remove operations may eliminate the need for the additional levels – ideally, we would like to keep the trie as compact as possible so that the subsequent lookup operations are faster. In Section 3.2 we show that removing an I-node that appears to be no longer needed may result in lost updates. We describe how to remove the keys while ensuring compression and no lost updates.

The Ctrie data structure is described in Figure 3. Each Ctrie contains a root reference to a so-called indirection node (I-node). An I-node contains a reference to a single node called a *main node*. There are several types of main nodes. A tomb node (T-node) is a special node used to ensure proper ordering during removals. A list node (L-node) is a leaf node used to handle hash code collisions by keeping such keys in a list. These are not immediately important, so we postpone discussion about T-nodes and L-nodes until Sections 3.2 and 3.3, respectively. A Ctrie node (C-node) is an internal main node containing a bitmap and the array with references to *branch nodes*. A branch node is either another I-node or a singleton node (S-node), which contains a single key and a value. S-nodes are leaves in the Ctrie (shown as key-value pairs in the figures).

The pseudocode in Figures 4, 6, 8, 9, 11, 13, 15 and 16 assumes short-circuiting semantics of the conditions in the *if* statements. We use logical symbols in boolean expressions. Pattern matching constructs match a node against its type and can be replaced with a sequence of *if-then-else* statements – we use pattern matching for conciseness. The colon (`:`) in the pattern matching cases should be read as *has type*. The keyword `def` denotes a procedure definition. Reads, writes and compare-and-set instructions written in capitals are atomic. This high level pseudocode might not be optimal in all cases – the source code contains a more efficient implementation.

The rest of the section describes the basic update operations.

3.1 Lookup and insert operations

A lookup starts by reading the root and then calls the recursive procedure `ilookup`, which traverses the Ctrie. This procedure either returns a result or a special value `RESTART`, which indicates that the lookup must be repeated.

The `ilookup` procedure reads the main node from the current I-node. If the main node is a C-node, then (as described in Section 3) the relevant bit `flag` of the bitmap and the index `pos` in the array are computed by the `flagpos` function. If the bitmap does not contain the relevant bit (line 10), then a key with the required

```
structure Ctrie {             structure CNode {
  root: INode                   bmp: integer
  readonly: boolean             array: Branch[2^W]
}                             }

structure Gen                 structure SNode {
                                k: KeyType
structure INode {               v: ValueType
  main: MainNode              }
  gen: Gen
}                             structure TNode {
                                sn: SNode
MainNode:                     }
  CNode | TNode | LNode
                              structure LNode {
Branch:                         sn: SNode
  INode | SNode                 next: LNode
                              }
```

Figure 3. Types and data structures

hashcode prefix is not present in the trie, so a NOTFOUND value is returned. Otherwise, the relevant branch at index pos is read from the array. If the branch is an I-node (line 12), the ilookup procedure is called recursively at the next level. If the branch is an S-node (line 14), then the key within the S-node is compared with the key being searched – these two keys have the same hashcode prefixes, but they need not be equal. If they are equal, the corresponding value from the S-node is returned and a NOTFOUND value otherwise. In all cases, the linearization point is the read in the line 7. This is because no nodes other than I-nodes change the value of their fields after they are created and we know that the main node was reachable in the trie at the time it was read in the line 7 [20].

If the main node within an I-node is a T-node (line 17), we try to remove it and convert it to a regular node before restarting the operation. This is described in more detail in Section 3.2. The L-node case is described in Section 3.3.

```
 1 def lookup(k)
 2   r = READ(root)
 3   res = ilookup(r, k, 0, null)
 4   if res ≠ RESTART return res else return lookup(k)
 5
 6 def ilookup(i, k, lev, parent)
 7   READ(i.main) match {
 8     case cn: CNode =>
 9       flag, pos = flagpos(k.hash, lev, cn.bmp)
10       if cn.bmp ⊙ flag = 0 return NOTFOUND
11       cn.array(pos) match {
12         case sin: INode =>
13           return ilookup(sin, k, lev + W, i)
14         case sn: SNode =>
15           if sn.k = k return sn.v else return NOTFOUND
16       }
17     case tn: TNode =>
18       clean(parent, lev - W)
19       return RESTART
20     case ln: LNode =>
21       return ln.lookup(k)
22   }
```

Figure 4. Lookup operation

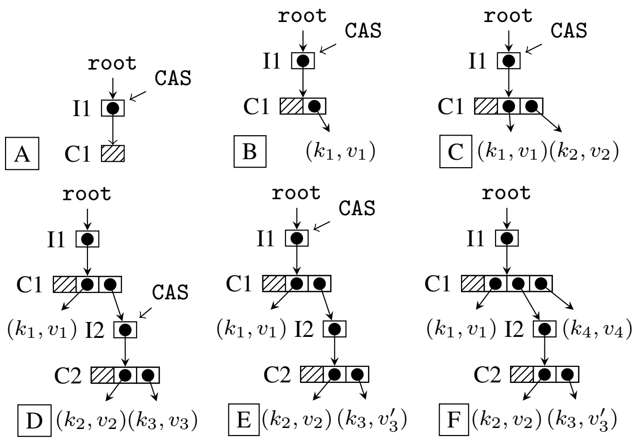

Figure 5. Ctrie insert

```
23 def insert(k, v)
24   r = READ(root)
25   if iinsert(r, k, v, 0, null) = RESTART insert(k, v)
26
27 def iinsert(i, k, v, lev, parent)
28   READ(i.main) match {
29     case cn: CNode =>
30       flag, pos = flagpos(k.hash, lev, cn.bmp)
31       if cn.bmp ⊙ flag = 0 {
32         ncn = cn.inserted(pos, flag, SNode(k, v))
33         if CAS(i.main, cn, ncn) return OK
34         else return RESTART
35       }
36       cn.array(pos) match {
37         case sin: INode =>
38           return iinsert(sin, k, v, lev + W, i)
39         case sn: SNode =>
40           if sn.k ≠ k {
41             nsn = SNode(k, v)
42             nin = INode(CNode(sn, nsn, lev + W))
43             ncn = cn.updated(pos, nin)
44             if CAS(i.main, cn, ncn) return OK
45             else return RESTART
46           } else {
47             ncn = cn.updated(pos, SNode(k, v))
48             if CAS(i.main, cn, ncn) return OK
49             else return RESTART
50           }
51       }
52     case tn: TNode =>
53       clean(parent, lev - W)
54       return RESTART
55     case ln: LNode =>
56       if CAS(i.main, ln, ln.inserted(k, v)) return OK
57       else return RESTART
58   }
```

Figure 6. Insert operation

When a new Ctrie is created, it contains a root I-node with the main node set to an empty C-node, which contains an empty bitmap and a zero-length array (Figure 5A). We maintain the invariant that only the root I-node can contain an empty C-node – all other C-nodes in the Ctrie contain at least one entry in their array. Inserting a key k_1 first reads the root and calling the procedure iinsert.

The procedure iinsert is invoked on the root I-node. This procedure works in a similar way as ilookup. If it reads a C-node within the I-node, it computes the relevant bit and the index in the array using the flagpos function. If the relevant bit is not in the bitmap (line 31) then a copy of the C-node with the new entry is created using the inserted function. The linearization point is a successful CAS in the line 33, which replaces the current C-node with a C-node containing the new key (see Figures 5A,B,C where two new keys k_1 and k_2 are inserted in that order starting from an empty Ctrie). An unsuccessful CAS means that some other operation already wrote to this I-node since its main node was read in the line 28, so the insert must be repeated.

If the relevant bit is present in the bitmap, then its corresponding branch is read from the array. If the branch is an I-node, then `iinsert` is called recursively. If the branch is an S-node and its key is not equal to the key being inserted (line 40), then the Ctrie has to be extended with an additional level. The C-node is replaced with its updated version (line 44), created using the `updated` function that adds a new I-node at the respective position. The new I-node has its main node pointing to a C-node with both keys. This scenario is shown in Figures 5C,D where a new key k_3 with the same hashcode prefix as k_2 is inserted. If the key in the S-node is equal to the key being inserted, then the C-node is replaced with its updated version with a new S-node. An example is given in the Figure 5E where a new S-node (k_3, v_3') replaces the S-node (k_3, v_3) from the Figure 5D. In both cases, the successful CAS instructions in the lines 44 and 48 are the linearization point.

Note that insertions to I-nodes at different levels may proceed concurrently, as shown in Figures 5E,F where a new key k_4 is added at the level 0, below the I-node $I1$. No race can occur, since the I-nodes at the lower levels remain referenced by the I-nodes at the upper levels even when new keys are added to the higher levels. This will not be the case after introducing the remove operation.

3.2 Remove operation

The remove operation has a similar control flow as the lookup and the insert operation. After examining the root, a recursive procedure `iremove` reads the main node of the I-node and proceeds casewise, removing the S-node from the trie by updating the C-node above it, similar to the insert operation.

The described approach has certain pitfalls. A remove operation may at one point create a C-node that has a single S-node below it. This is shown in Figure 7A, where the key k_2 is removed from the Ctrie. The resulting Ctrie in Figure 7B is still valid in the sense that the subsequent insert and lookup operations will work. However, these operations could be faster if (k_3, v_3) were moved into the C-node below $I1$. After having removed the S-node (k_2, v_2), the remove operation could create an updated version of $C1$ with a reference to the S-node (k_3, v_3) instead of $I2$ and write that into $I1$ to compress the Ctrie. But, if a concurrent insert operation were to write to $I2$ just before $I1$ was updated with the compressed version of $C1$, the insertion would be lost.

To solve this problem, we introduce a new type of a main node called a tomb node (T-node). We introduce the following invariant to Ctries – if an I-node points to a T-node at some time t_0 then for all times greater than t_0, the I-node points to the same T-node. In other words, a T-node is the last value assigned to an I-node. This ensures that no inserts occur at an I-node if it is being compressed. An I-node pointing to a T-node is called a *tombed I-node*.

The remove operation starts by reading the root I-node and calling the recursive procedure `iremove`. If the main node is a C-node, the `flagpos` function is used to compute the relevant bit and the branch position. If the bit is not present in the bitmap (line 69), then a `NOTFOUND` value is returned. In this case, the linearization point is the read in the line 66. Otherwise, the branch node is read from the array. If the branch is another I-node, the procedure is called recursively. If the branch is an S-node, its key is compared against the key being removed. If the keys are not equal (line 75), the `NOTFOUND` value is returned and the linearization point is the read in the line 66. If the keys are equal, a copy of the current node without the S-node is created. The *contraction* of the copy is then created using the `toContracted` procedure. A successful CAS in the line 79 will substitute the old C-node with the copied C-node, thus removing the S-node with the given key from the trie – this is the linearization point.

If a given C-node has only a single S-node below and is not at the root level (line 101) then the `toContracted` procedure returns a T-node that wraps the S-node. Otherwise, it just returns the given C-node. This ensures that every I-node except the root points to a C-node with at least one branch. Furthermore, if it points to exactly one branch, then that branch is not an S-node (this scenario is possible if two keys with the same hashcode prefixes are inserted). Calling this procedure ensures that the CAS in the line 79 replaces the C-node $C2$ from the Figure 7A with the T-node in Figure 7C instead of the C-node $C2$ in Figure 7B. This CAS is the linearization point since the S-node (k_2, v_2) is no longer in the trie. However, it does not solve the problem of compressing the Ctrie (we ultimately want to obtain a Ctrie in Figure 7D). In fact,

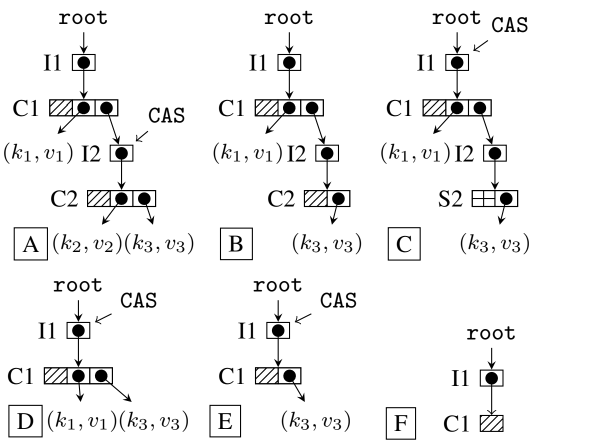

Figure 7. Ctrie remove

```
59 def remove(k)
60   r = READ(root)
61   res = iremove(r, k, 0, null)
62   if res ≠ RESTART return res
63   else return remove(k)
64
65 def iremove(i, k, lev, parent)
66   READ(i.main) match {
67     case cn: CNode =>
68       flag, pos = flagpos(k.hash, lev, cn.bmp)
69       if cn.bmp ⊙ flag = 0 return NOTFOUND
70       res = cn.array(pos) match {
71         case sin: INode =>
72           iremove(sin, k, lev + W, i)
73         case sn: SNode =>
74           if sn.k ≠ k
75             NOTFOUND
76           else {
77             ncn = cn.removed(pos, flag)
78             cntr = toContracted(ncn, lev)
79             if CAS(i.main, cn, cntr) sn.v else RESTART
80           }
81       }
82       if res = NOTFOUND ∨ res = RESTART return res
83       if READ(i.main): TNode
84         cleanParent(parent, in, k.hash, lev - W)
85       return res
86     case tn: TNode =>
87       clean(parent, lev - W)
88       return RESTART
89     case ln: LNode =>
90       nln = ln.removed(k)
91       if length(nln) = 1 nln = entomb(nln.sn)
92       if CAS(i.main, ln, nln) return ln.lookup(k)
93       else return RESTART
94   }
```

Figure 8. Remove operation

given a Ctrie containing two keys with long matching hashcode prefixes, removing one of these keys will create a arbitrarily long chain of C-nodes with a single T-node at the end. We introduced the invariant that no tombed I-node changes its main node. To remove the tombed I-node, the reference to it in the C-node above must be changed with a reference to its *resurrection*. A resurrection of a tombed I-node is the S-node wrapped in its T-node. For all other branch nodes, the resurrection is the node itself.

To ensure compression, the remove operation checks if the current main node is a T-node after removing the key from the Ctrie (line 83). If it is, it calls the `cleanParent` procedure, which reads the main node of the parent I-node `p` and the current I-node `i` in the line 113. It then checks if the T-node below `i` is reachable from `p`. If `i` is no longer reachable, then it returns – some other thread must have already completed the contraction. If it is reachable then it replaces the C-node below `p`, which contains the tombed I-node `i` with a copy updated with the resurrection of `i` (CAS in the line 122). This copy is possibly once more contracted into a T-node at a higher level by the `toContracted` procedure.

```
 95 def toCompressed(cn, lev)
 96   num = bit#(cn.bmp)
 97   ncn = cn.mapped(resurrect(_))
 98   return toContracted(ncn, lev)
 99
100 def toContracted(cn, lev)
101   if lev > 0 $\wedge$ cn.array.length = 1
102     cn.array(0) match {
103       case sn: SNode => return entomb(sn)
104       case _ => return cn
105     }
106   else return cn
107
108 def clean(i, lev)
109   m = READ(i.main)
110   if m: CNode CAS(i.main, m, toCompressed(m, lev))
111
112 def cleanParent(p, i, hc, lev)
113   m, pm = READ(i.main), READ(p.main)
114   pm match {
115     case cn: CNode =>
116       flag, pos = flagpos(k.hash, lev, cn.bmp)
117       if bmp $\odot$ flag = 0 return
118       sub = cn.array(pos)
119       if sub $\neq$ i return
120       if m: TNode {
121         ncn = cn.updated(pos, resurrect(m))
122         if $\neg$CAS(p.main, cn, toContracted(ncn, lev))
123           cleanParent(p, i, hc, lev)
124       }
125     case _ => return
126   }
```

Figure 9. Compression operations

To preserve the lock-freedom property, all operations that read a T-node must help compress it instead of waiting for the removing thread to complete the compression. For example, after finding a T-node lookups call the `clean` procedure on the parent node in the line 17. This procedure creates the *compression* of the given C-node – a new C-node with all the tombed I-nodes below resurrected. This new C-node is contracted if possible. The old C-node is then replaced with its compression with the CAS in the line 110. Note that neither `clean` nor `cleanParent` are ever called for the parent of the root, since the root never contains a T-node. For example, removing the S-node (k_3, v_3) from the Ctrie in Figure 7D produces a Ctrie in Figure 7E. A subsequent remove produces an empty trie in Figure 7F.

Both insert and lookup are tail-recursive and may be rewritten to loop-based variants, but this is not so trivial with the remove operation. Since remove operations must be able to compress arbitrary long chains of C-nodes, the call stack is used to store information about the path in the Ctrie being traversed.

3.3 Hash collisions

In this implementation, hash tries use a 32-bit hashcode space. Although hash collisions are rare, it is still possible that two unequal keys with the same hashcodes are inserted. To preserve correctness, we introduce a new type of nodes called list nodes (L-nodes), which are basically persistent linked lists. If two keys with the same hashcodes collide, we place them inside an L-node.

We add another case to the basic operations from Section 3. Persistent linked list operations `lookup`, `inserted`, `removed` and `length` are trivial and not included in the pseudocode. We additionally check if the updated L-node in the `iremove` procedure has length 1 and replace the old node with a T-node in this case.

Another important change is in the `CNode` constructor in line 42. This constructor was a recursive procedure that creates additional C-nodes as long as the hashcode chunks of the two keys are equal at the given level. We modify it to create an L-node if the level is greater than the length of the hashcode – in our case 32.

3.4 Additional operations

Collection classes in various frameworks typically need to implement additional operations. For example, the `ConcurrentMap` interface in Java defines four additional methods: `putIfAbsent`, `replace` any value a key is mapped to with a new value, `replace` a specific value a key is mapped to with a new value and `remove` a key mapped to a specific value. All of these operations can be implemented with trivial modifications to the operations introduced in Section 3. For example, removing a key mapped to a specific value can be implemented by adding an additional check `sn.v = v` to the line 74. We invite the reader to try to inspect the source code of our implementation.

Methods such as `size`, `iterator` or `clear` commonly seen in collection frameworks cannot be implemented in a lock-free, linearizable manner so easily. The reason for this is that they require global information about the data structure at one specific instance in time – at first glance, this requires locking or weakening the contract so that these methods can only be called during a quiescent state.

It turns out that for Ctries these methods can be computed efficiently and correctly by relying on a constant time lock-free, atomic snapshot.

4. Snapshot

While creating a consistent snapshot often seems to require copying all of the elements of a data structure, this is not generally the case. Persistent data structures present in functional languages have operations that return their updated versions and avoid copying all the elements, typically achieving logarithmic or sometimes even constant complexity [19].

A persistent hash trie data structure seen in standard libraries of languages like Scala or Clojure is updated by rewriting the path from the root of the hash trie to the leaf the key belongs to, leaving the rest of the trie intact. This idea can be applied to implement the snapshot. A generation count can be assigned to each I-node. A snapshot is created by copying the root I-node and setting it to the new generation. When some update operation detects that an I-node being read has a generation older than the generation of the root, it can create a copy of that I-node initialized with the latest generation and update the parent accordingly – the effect of this is that after the snapshot is taken, a path from the root to some leaf is updated only the first time it is accessed, analogous to persistent data structures. The snapshot is thus an $O(1)$ operation, while all other operations preserve an $O(\log n)$ complexity, albeit with a larger constant factor.

Still, the snapshot operation will not work as described above, due to the races between the thread creating the snapshot and threads that have already read the root I-node with the old generation and are traversing the Ctrie in order to update it. The problem is that a CAS that is a linearization point for an insert (e.g. in the line 48) can be preceeded by the snapshot creation – ideally, we want such a CAS instruction to fail, since the generation of the Ctrie root has changed. If we used a DCAS instruction instead, we could ensure that the write occurs only if the Ctrie root generation remained the same. However, most platforms do not support an efficient implementation of this instruction yet. On closer inspection, we find that an RDCSS instruction described by Harris et al. [10] that does a double compare and a single swap is enough to implement safe updates. The downside of RDCSS is that its software implementation creates an intermediate descriptor object. While such a construction is general, due to the overhead of allocating and later garbage collecting the descriptor, it is not optimal in our case.

We will instead describe a new procedure called generation-compare-and-swap, or GCAS. This procedure has semantics similar to that of the RDCSS, but it does not create the intermediate object except in the case of failures that occur due to the snapshot being taken – in this case the number of intermediate objects created per snapshot is $O(t)$ where t is the number of threads invoking some Ctrie operation at the time.

4.1 GCAS procedure

The `GCAS` procedure has the following preconditions. It takes 3 parameters – an I-node `in`, and two main nodes `old` and `n`. Only the thread invoking the `GCAS` procedure may have a reference to the main node `n`[1]. Each main node must contain an additional field `prev` that is not accessed by the clients. Each I-node must contain an additional immutable field `gen`. The `in.main` field is only read using the `GCAS_READ` procedure.

```
def GCAS(in, old, n)
  r = READ(in.main)
  if r = old ∧ in.gen = READ(root).gen {
    WRITE(in.main, n)
    return ⊤
  } else return ⊥
```

Figure 10. GCAS semantics

Its semantics are equivalent to an atomic block shown in Figure 10. The `GCAS` is similar to a CAS instruction with the difference that it also compares if I-node `gen` field is equal to the `gen` field of the Ctrie root. The `GCAS` instruction is also lock-free. We show the implementation in Figure 11, based on single-word CAS instructions. The idea is to communicate the intent of replacing the value in the I-node and check the generation field in the root before committing to the new value.

The `GCAS` procedure starts by setting the `prev` field in the new main node `n` to point at main node `old`, which will be the expected value for the first CAS. Since the preconditions state that no other thread sees `n` at this point, this write is safe. The thread proceeds by proposing the new value `n` with a CAS instruction in line 129. If the CAS fails then `GCAS` returns ⊥ and the CAS is the linearization point. If the CAS succeeds (shown in Figure 12B), the new main node is not yet committed – the generation of the root has to be

[1] This is easy to ensure in environments with automatic memory management and garbage collection. Otherwise, a technique similar to the one proposed by Herlihy [11] can be used to ensure that a thread does not reuse objects that have already been recycled.

```
127 def GCAS(in, old, n)
128   WRITE(n.prev, old)
129   if CAS(in.main, old, n) {
130     GCAS_Commit(in, n)
131     return READ(n.prev) = null
132   } else return ⊥
133
134 def GCAS_Commit(in, m)
135   p = READ(m.prev)
136   r = ABORTABLE_READ(root)
137   p match {
138     case n: MainNode =>
139       if (r.gen = in.gen ∧ ¬readonly) {
140         if CAS(m.prev, p, null) return m
141         else return GCAS_Commit(in, m)
142       } else {
143         CAS(m.prev, p, new Failed(p))
144         return GCAS_Commit(in, READ(in.main))
145       }
146     case fn: Failed =>
147       if CAS(in.main, m, fn.prev) return fn.prev
148       else return GCAS_Commit(in, READ(in.main))
149     case null => return m
150   }
151
152 def GCAS_READ(in)
153   m = READ(in.main)
154   if (READ(m.prev) = null) return m
155   else return GCAS_Commit(in, m)
```

Figure 11. GCAS operations

compared against the generation of the I-node before committing the value, so the tail-recursive procedure `GCAS_Commit` is called with the parameter `m` set to the proposed value `n`. This procedure reads the previous value of the proposed node and the Ctrie root. We postpone the explanation of the `ABORTABLE_READ` procedure until Section 4.2 – for now it can be considered an ordinary atomic `READ`. It then inspects the previous value.

If the previous value is a main node different than `null`, the root generation is compared to the I-node generation. If the generations are equal, the `prev` field in `m` must be set to `null` to complete the `GCAS` (Figure 12C). If the CAS in the line 140 fails, the procedure is repeated. If the generations are not equal, the `prev` field is set to a special `Failed` node whose previous value is set to `m.prev` (Figure 12D), and the `GCAS_Commit` procedure is repeated. This special node signals that the `GCAS` has failed and that the I-node main node must be set back to the previous value.

If the previous value is a failed node, then the main node of the I-node is set back to the previous value from the failed node by the CAS in the line 147 (Figure 12D,E). If the CAS is not successful, the procedure must be repeated after rereading the main node.

If the previous value is `null`, then some other thread already checked the generation of the root and committed the node, so the method just returns the current node.

Once the `GCAS_Commit` procedure returns, `GCAS` checks if the `prev` field of the proposed node is `null`, meaning that the value had been successfully committed at some point.

If the proposed value is rejected, the linearization point is the CAS in line 147, which sets the main node of an I-node back to the previous value (this need not necessarily be done by the current thread). If the proposed value is accepted, the linearization point is the successful CAS in the line 140 – independent of that CAS was done by the current thread or some other thread. If the linearization point is external, we know it happened after `GCAS` was invoked. We know that the `gen` field does not change during the lifetime of an I-node, so it remains the same until a successful CAS in the line 140. If some other thread replaces the root with a new I-node with a different `gen` field after the read in the line 136, then no other thread

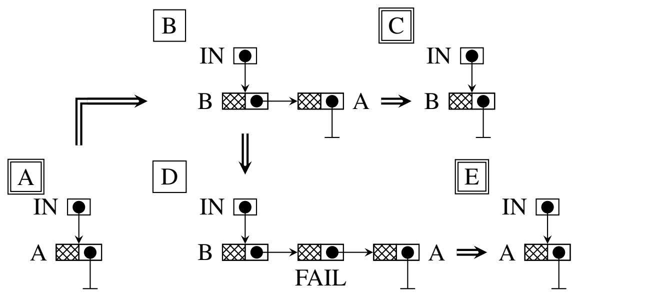

Figure 12. GCAS states

that observed the root change will succeed in writing a failed node, since we assumed that the CAS in the line 140 succeeded.

To ensure lock-freedom, the `GCAS_READ` procedure must help commit if it finds a proposed value. After reading the main node, it checks if its `prev` field is set to `null`. If it is, it can safely return the node read in line 153 (which is the linearization point) since the algorithm never changes the `prev` field of a committed node and comitting a node sets the `prev` field to `null`. If `prev` is different than `null`, the node hasn't been committed yet, so `GCAS_Commit` is called to complete the read. In this case, the value returned by `GCAS_Commit` is the result and the linearization points are the same as with `GCAS` invoking the `GCAS_Commit`.

Both `GCAS` and `GCAS_READ` are designed to add a non-significant amount of overhead compared a single CAS instruction and a read, respectively. In particular, if there are no concurrent modifications, a `GCAS_READ` amounts to an atomic read of the node, an atomic read of its `prev` field, a comparison and a branch.

4.2 Implementation

We now show how to augment the existing algorithm with snapshots using the `GCAS` and `GCAS_READ` procedures. We add a `prev` field to each type of a main node and a `gen` field to I-nodes. The `gen` field points to generation objects allocated on the heap. We do not use integers to avoid overflows and we do not use pointers to the root as generation objects, since that could cause memory leaks – if we did, the Ctrie could potentially transitively point to all of its previous snapshot versions. We add an additional parameter `startgen` to procedures `ilookup`, `iinsert` and `iremove`. This parameter contains the generation count of the Ctrie root, which was read when the operation began.

Next, we replace every occurence of a CAS instruction with a call to the `GCAS` procedure. We replace every atomic read with a call to the `GCAS_READ` procedure. Whenever we read an I-node while traversing the trie (lines 12, 37 and 71) we check if the I-node generation corresponds to `startgen`. If it does, we proceed as before. Otherwise, we create a copy of the current C-node such that all of its I-nodes are copied to the newest generation and use `GCAS` to update the main node before revisiting the current I-node again. This is shown in Figure 13, where the `cn` refers to the C-node currently in scope (see Figures 4, 6 and 8). In line 43 we copy the C-node so that all I-nodes directly below it are at the latest generation before updating it. The `readonly` field is used to check if the Ctrie is read-only - we explain this shortly. Finally, we add a check to the `cleanParent` procedure, which aborts if `startgen` is different than the `gen` field of the I-node.

All `GCAS` invocations fail if the generation of the Ctrie root changes and these failures cause the basic operations to be restarted. Since the root is read once again after restarting, we are guaranteed to restart the basic operation with the updated value of the `startgen` parameter.

```
...
  case sin: INode =>
    if (startgen eq in.gen)
      return iinsert(sin, k, v, lev + W, i, startgen)
    else
      if (GCAS(cn, atGen(cn, startgen)))
        iinsert(i, k, v, lev, parent, startgen)
      else return RESTART
...
def atGen(n, ngen)
  n match {
    case cn: CNode => cn.mapped(atGen(_, ngen))
    case in: INode => new INode(GCAS_READ(in), ngen)
    case sn: SNode => sn
  }
```

Figure 13. I-node renewal

One might be tempted to implement the snapshot operation by simply using a CAS instruction to change the `root` reference of a Ctrie to point to an I-node with a new generation. However, the snapshot operation can copy and replace the root I-node only if its main node does not change between the copy and the replacement.

We use the `RDCSS` procedure described by Harris et al. [10], which works in a similar way as `GCAS`, but proposes the new value by creating an intermediate descriptor object, which points to the previous and the proposed value. We do not see the cost of allocating it as critical since we expect a snapshot to occur much less often than the other update operations. We specialize `RDCSS` – the first compare is on the root and the second compare is always on the main node of the old value of the root. `GCAS_READ` is used to read the main node of the old value of the root. The semantics correspond to the atomic block shown in Figure 14.

```
def RDCSS(ov, ovmain, nv)
  r = READ(root)
  if r = ov ∧ GCAS_READ(ov.main) = ovmain {
    WRITE(root, nv)
    return ⊤
  } else return ⊥
```

Figure 14. Modified RDCSS semantics

To create a snapshot of the Ctrie the root I-node is read. Its main node is read next. The `RDCSS` procedure is called, which replaces the old root I-node with its new generation copy. If the `RDCSS` is successful, a new Ctrie is returned with the copy of the root I-node set to yet another new generation. Otherwise, the snapshot operation is restarted.

```
def snapshot()
  r = RDCSS_READ()
  expmain = GCAS_READ(r)
  if RDCSS(r, expmain, new INode(expmain, new Gen))
    return new Ctrie {
      root = new INode(expmain, new Gen)
      readonly = ⊥
    }
  else return snapshot()
```

Figure 15. Snapshot operation

An observant reader will notice that if two threads simultaneously start a `GCAS` on the root I-node and an `RDCSS` on the root field of the Ctrie, the algorithm will deadlock[2] since both locations contain the proposed value and read the other location before committing. To avoid this, one of the operations has to have a higher

[2] Actually, it will cause a stack overflow in the current implementation.

priority. This is the reason for the `ABORTABLE_READ` in line 136 in Figure 11 – it is a modification of the `RDCSS_READ` that writes back the old value to the root field if it finds the proposal descriptor there, causing the `snapshot` to be restarted. The algorithm remains lock-free, since the `snapshot` reads the main node in the root I-node before restarting, thus having to commit the proposed main node.

Since both the original Ctrie and the snapshot have a root with a new generation, both Ctries will have to rebuild paths from the root to the leaf being updated. When computing the size of the Ctrie or iterating the elements, we know that the snapshot will not be modified, so updating paths from the root to the leaf induces an unnecessary overhead. To accomodate this we implement the `readOnlySnapshot` procedure that returns a read only snapshot. The only difference with respect to the `snapshot` procedure in Figure 15 is that the returned Ctrie has the old root `r` (line 167) and the `readonly` field is set to $\top$. The `readonly` field mentioned earlier in Figures 3, 11 and 13 guarantees that no writes to I-nodes occur if it is set to $\top$. This means that paths from the root to the leaf being read are not rewritten in read-only Ctries. The rule also applies to T-nodes – instead of trying to clean the Ctrie by resurrecting the I-node above the T-node, the lookup in a read-only Ctrie treats the T-node as if it were an S-node. Furthermore, if the `GCAS_READ` procedure tries to read from an I-node in which a value is proposed, it will abort the write by creating a failed node and then writing the old value back (line 139).

```
def iterator()
  if readonly return new Iterator(RDCSS_READ(root))
  else return readOnlySnapshot().iterator()

def size()
  sz, it = 0, iterator()
  while it.hasNext sz += 1
  return sz

def clear()
  r = RDCSS_READ()
  expmain = GCAS_READ(r)
  if ¬RDCSS(r, expmain, new INode(new Gen)) clear()
```

Figure 16. Snapshot-based operations

Finally, we show how to implement snapshot-based operations in Figure 16. The `size` operation can be optimized further by caching the size information in main nodes of a read-only Ctrie – this reduces the amortized complexity of the `size` operation to $O(1)$ because the size computation is amortized across the update operations that occurred since the last snapshot. For reasons of space, we do not go into details nor do we show the entire iterator implementation, which is trivial once a snapshot is obtained.

5. Evaluation

We performed experimental measurements on a JDK6 configuration with a quad-core 2.67 GHz Intel i7 processor with 8 hyperthreads, a JDK6 configuration with an 8-core 1.165 GHz Sun UltraSPARC-T2 processor with 64 hyperthreads and a JDK7 configuration with four 8-core Intel Xeon 2.27 GHz processors with a total of 64 hyperthreads. The first configuration has a single multicore processor, the second has a single multicore processor, but a different architecture and the third has several multicore processors on one motherboard. We followed established performance measurement methodologies [9]. We compared the performance of the Ctrie data structure against the `ConcurrentHashMap` and the `ConcurrentSkipListMap` from the Java standard library, as well as the Cliff Click's non-blocking concurrent hash map implementation [5]. All of the benchmarks show the number of threads used on the x-axis and the throughput on the y-axis. In all experiments, the Ctrie supports the snapshot operation.

The first benchmark called $insert$ starts with an empty data structure and inserts $N = 1000000$ entries into the data structure. The work of inserting the elements is divided equally between P threads, where P varies between 1 and the maximum number of hyperthreads on the configuration (x-axis). The y-axis shows throughput – the number of times the benchmark is repeated per second. This benchmark is designed to test the scalability of the resizing, since the data structure is initially empty. Data structures like hash tables, which have a resize phase, do no seem to be very scalable for this particular use-case, as shown in Figure 17. On the Sun UltraSPARC-T2 (Figure 18), the Java concurrent hash map scales for up to 4 threads. Cliff Click's nonblocking hash table scales, but the cost of the resize is so high that this is not visible on the graph. Concurrent skip lists scale well in this test, but Ctries are a clear winner here since they achieve an almost linear speedup for up to 32 threads and an additional speedup as the number of threads reaches 64.

The benchmark $lookup$ does $N = 1000000$ lookups on a previously created data structure with N elements. The work of looking up all the elements is divided between P threads, where P varies as before. Concurrent hash tables perform especially well in this benchmark on all three configurations – the lookup operation mostly amounts to an array read, so the hash tables are $2-3$ times faster than Ctries. Ctries, in turn, are faster than skip lists due to a lower number of indirections, resulting in fewer cache misses.

The $remove$ benchmark starts with a previously created data structure with $N = 1000000$ elements. It removes all of the elements from the data structure. The work of removing all the elements is divided between P threads, where P varies. On the quad-core processor (Figure 17) both the Java concurrent skip list and the concurrent hash table scale, but not as fast as Ctries or the Cliff Click's nonblocking hash table. On the UltraSPARC-T2 configuration (Figure 18), the nonblocking hash table is even up to 2.5 times faster than Ctries. However, we should point out that the nonblocking hash table does not perform compression – once the underlying table is resized to a certain size, the memory is used regardless of whether the elements are removed. This can be a problem for long running applications and applications using a greater number of concurrent data structures.

The next three benchmarks called $90-9-1$, $80-15-5$ and $60-30-10$ show the performance of the data structures when the operations are invoked in the respective ratio. Starting from an empty data structure, a total of $N = 1000000$ invocations are done. The work is divided equally among P threads. For the $90-9-1$ ratio the Java concurrent hash table works very well on both the quad-core configuration and the UltraSPARC-T2. For the $60-30-10$ ratio Ctries seem to do as well as the nonblocking hash table. Interestingly, Ctries seem to outperform the other data structures in all three tests on the 4x 8-core i7 (Figure 19).

The $preallocated-5-4-1$ benchmark in Figure 18 proceeds exactly as the previous three benchmarks with the difference that it starts with a data structure that contains all the elements. The consequence is that the hash tables do not have to be resized – this is why the Java concurrent hash table performs better for P up to 16, but suffers a performance degradation for bigger P. For $P > 32$ Ctries seem to do better. In this benchmarks, the nonblocking hash table was 3 times faster than the other data structures, so it was excluded from the graph. For applications where the data structure size is known in advance this may be an ideal solution – for others, preallocating may result in a waste of memory.

To evaluate snapshot performance, we do 2 kinds of benchmarks. The $snapshot-remove$ benchmark in Figure 20 is similar to the $remove$ benchmark – it measures the performance of removing all the elements from a snapshot of a Ctrie and compares that time to removing all the elements from an ordinary Ctrie. On both i7 configurations (Figures 17 and 19), removing from a snapshot is up to 50% slower, but scales in the same way as removing from an ordinary Ctrie. On the UltraSPARC-T2 configuration (Figure 18), this gap is much smaller. The benchmark $snapshot - lookup$ in Figure 21 is similar to the last one, with the difference that all the elements are looked up once instead of being removed. Looking up elements in the snapshot is slower, since the Ctrie needs to be fully reevaluated. Here, the gap is somewhat greater on the UltraSPARC-T2 configuration and smaller on the i7 configurations.

Finally, the $PageRank$ benchmark in Figure 22 compares the performance of iterating parts of the snapshot in parallel against the performance of filtering out the page set in each iteration as explained in Section 2. The snapshot-based implementation is much faster on the i7 configurations, whereas the difference is not that much pronounced on the UltraSPARC-T2.

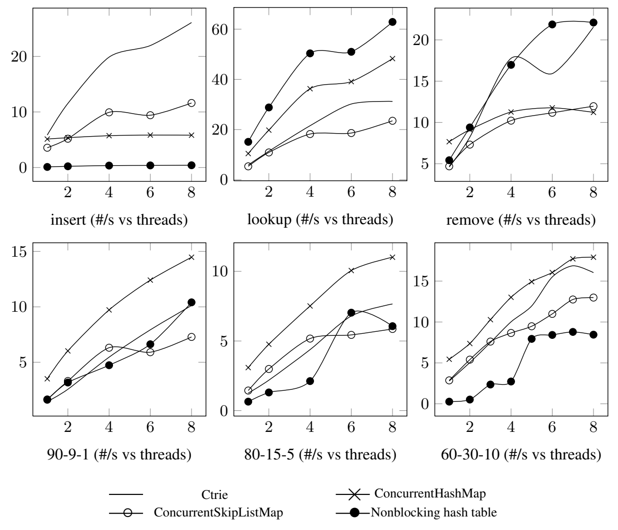

Figure 17. Basic operations, quad-core i7

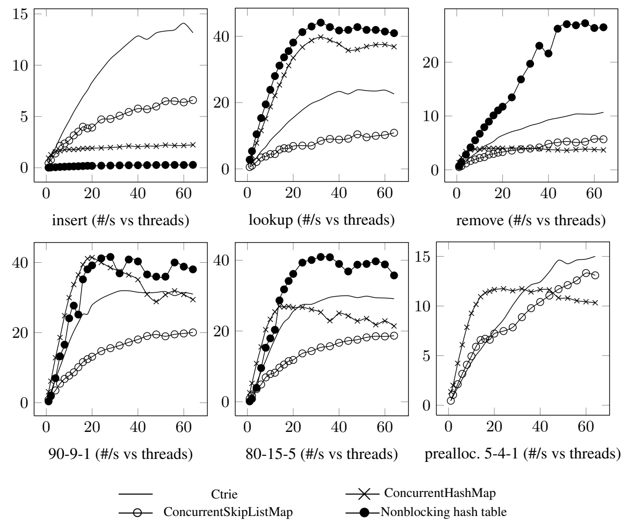

Figure 18. Basic operations, 64 hyperthread UltraSPARC-T2

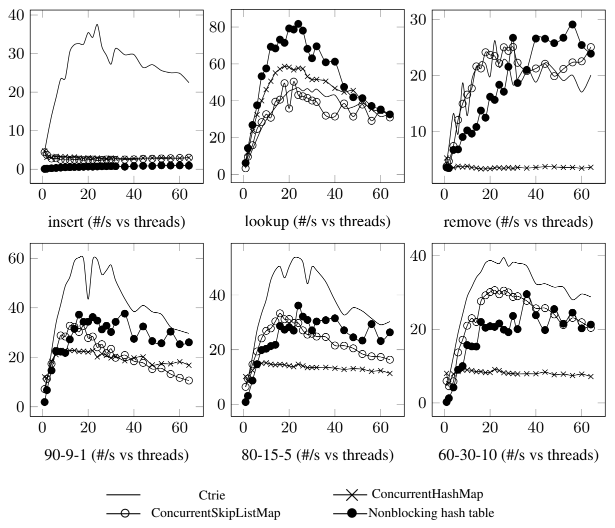

Figure 19. Basic operations, 4x 8-core i7

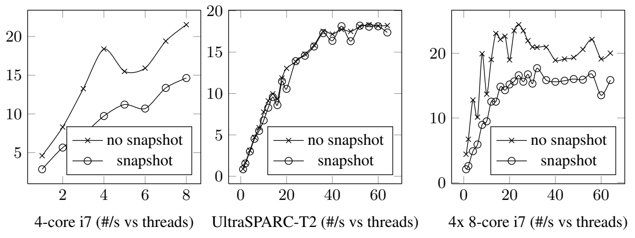

Figure 20. Remove vs. snapshot remove

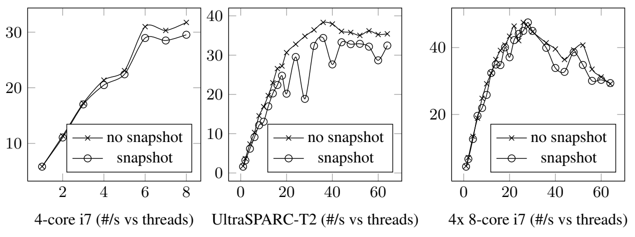

Figure 21. Lookup vs. snapshot lookup

6. Related work

Moir and Shavit give an overview of concurrent data structures [18]. A lot of research was done on concurrent lists, queues and concurrent priority queues. Linked lists are an inefficient implementation of a map abstraction because they do not scale well and the latter two do not support the basic map operations.

Hash tables are resizeable arrays of buckets. Each bucket holds some number of elements that is expected to be constant. The constant number of elements per bucket requires resizing the data structure as more elements are added – sequential hash tables amortize the resizing cost the table over other operations [6]. While the individual concurrent hash table operations such as insertion or removal can be performed in a lock-free manner as shown by Maged [17], resizing is typically implemented with a global lock. Although the cost of resizing is amortized against operations by

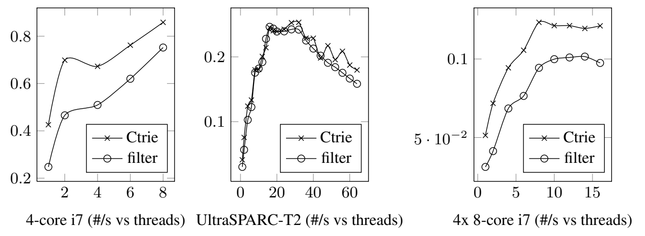

Figure 22. Pagerank

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 [14]. Shalev and Shavit give an innovative approach to resizing – split-ordered lists keep a table of hints into a single linked list in a way that does not require rearranging the elements of the linked list when resizing the table [25].

Skip lists store 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 [23]. Pugh proposed concurrent skip lists that achieve synchronization using locks [22]. Concurrent non-blocking skip lists were later implemented by Lev, Herlihy, Luchangco and Shavit [12] and Lea [14].

Concurrent binary search trees were proposed by Kung and Lehman [13] – 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 [4]. Recently, the first non-blocking implementation of a binary search tree was proposed [7].

Tries were originally proposed by Brandais [3] and Fredkin [8]. Trie hashing was applied to accessing files stored on the disk by Litwin [15]. Litwin, Sagiv and Vidyasankar implemented trie hashing in a concurrent setting [16], however, they did so by using mutual exclusion locks. Hash array mapped trees, or hash tries, are tries for shared-memory described by Bagwell [2]. To our knowledge, there is no nonblocking concurrent implementation of hash tries prior to our work. A correctness proof for the basic operations is presented as part of our tech report on the version of concurrent hash tries that do not support lock-free snapshots [20].

A persistent data structure is a data structure that preserves its previous version when being modified. Efficient persistent data structures are in use today that re-evaluate only a small part of the data structure on modification, thus typically achieving logarithmic, amortized constant and even constant time bounds for their operations. Okasaki presents an overview of persistent data structures [19]. Persistent hash tries have been introduced in standard libraries of languages like Scala [24] and Clojure.

`RDCSS` and `DCAS` software implementations have been described by Harris [10]. In the past, `DCAS` has been used to implement lock-free concurrent deques [1].

7. Conclusion

We described a concurrent implementation of the hash trie data structure with lock-free update operations. We described an $O(1)$ lock-free, atomic snapshot operation that allows efficient traversal, amortized $O(1)$ size retrieval and an $O(1)$ clear operation. It is apparent that a hardware supported DCAS instruction would simplify the design of both update and the snapshot operations.

As a future work direction, we postulate that the GCAS approach can be applied to other data structures – given a CAS-based concurrent data structure and its persistent sequential version, a lock-free, atomic snapshot operation can be implemented by adding a generation counter fields to nodes in the data structure and replacing the CAS instructions that are linearization points with GCAS instructions. This approach appears to be particularly applicable to tree-based data structures.

Acknowledgments

We would like to express our thanks to prof. Doug Lea at the State University of New York at Oswego for his help. We would also like to thank Eva Darulova and Tihomir Gvero at the EPFL for the useful discussions we've lead.

References

[1] O. Agesen, D. L. Detlefs, C. H. Flood, A. Garthwaite, P. A. Martin , N. Shavit , G. L. Steele Jr.: DCAS-Based Concurrent Deques. SPAA, 2000.

[2] P. Bagwell: Ideal Hash Trees. EPFL Technical Report, 2001.

[3] R. Brandais: File searching using variable length keys. Proceedings of Western Joint Computer Conference, 1959.

[4] N. G. Bronson, J. Casper, H. Chafi, K. Olukotun: A Practical Concurrent Binary Search Tree. Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2009.

[5] C. Click: Towards a Scalable Non-Blocking Coding Style. http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms, 2nd Edition. The MIT Press, 2001.

[7] F. Ellen, P. Fatourou, E. Ruppert, F. van Breugel: Non-blocking binary search trees. PODC, 2010.

[8] E. Fredkin: Trie memory. Communications of the ACM, 1960.

[9] A. Georges, D. Buytaert, L. Eeckhout: Statistically Rigorous Java Performance Evaluation. OOPSLA, 2007.

[10] T. L. Harris, K. Fraser, I. A. Pratt: A Practical Multi-word Compare-and-Swap Operation. DISC, 2002.

[11] M. Herlihy: A Methodology for Implementing Highly Concurrent Data Structures. PPOPP, 1990.

[12] M. Herlihy, Y. Lev, V. Luchangco, N. Shavit: A Provably Correct Scalable Concurrent Skip List. OPODIS, 2006.

[13] H. Kung, P. Lehman: Concurrent manipulation of binary search trees. ACM Transactions on Database Systems (TODS), vol. 5, issue 3, 1980.

[14] Doug Lea's Home Page: http://gee.cs.oswego.edu/

[15] W. Litwin: Trie Hashing. Proceedings of the 1981 ACM SIGMOD international conference on Management of data, 1981.

[16] W. Litwin, Y. Sagiv, K. Vidyasankar: Concurrency and Trie Hashing. Acta Informatica archive, vol. 26, issue 7, 1989.

[17] Maged M. Michael: High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. SPAA, 2002.

[18] M. Moir, N. Shavit: Concurrent data structures. Handbook of Data Structures and Applications, Chapman and Hall, 2004.

[19] C. Okasaki: Purely Functional Data Structures. Cambridge University Press, 1999.

[20] A. Prokopec, P. Bagwell, M. Odersky: Cache-Aware Lock-Free Concurrent Hash Tries. EPFL Technical Report, 2011.

[21] A. Prokopec, P. Bagwell, T. Rompf, M. Odersky, A Generic Parallel Collection Framework. Euro-Par 2011 Parallel Processing, 2011.

[22] William Pugh: Concurrent Maintenance of Skip Lists. UM Technical Report, 1990.

[23] William Pugh: Skip Lists: A Probabilistic Alternative to Balanced Trees. Communications ACM, volume 33, 1990.

[24] The Scala Programming Language Homepage. http://www.scala-lang.org/

[25] O. Shalev, N. Shavit: Split-Ordered Lists: Lock-Free Extensible Hash Tables. Journal of the ACM, vol. 53., no. 3., 2006.