Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Add links

Ctrie

From Wikipedia, the free encyclopedia

Not to be confused with C-trie.

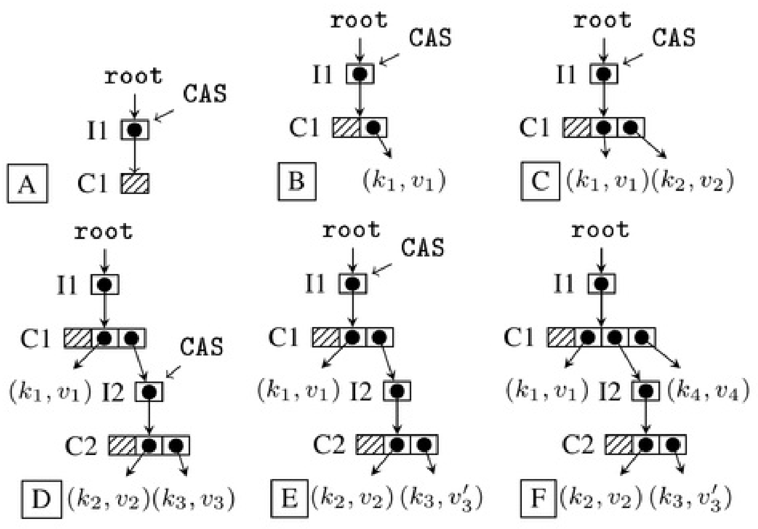
A **concurrent hash-trie** or **Ctrie**[1][2] is a concurrent thread-safe lock-free implementation of a hash array mapped trie. It is used to implement the concurrent map abstraction. It has particularly scalable concurrent insert and remove operations and is memory-efficient.[3] It is the first known concurrent data-structure that supports O(1), atomic, lock-free snapshots.[2][4]

Contents [hide]

Operation [edit]

The Ctrie data structure is a non-blocking concurrent hash array mapped trie based on single-word compare-and-swap instructions in a shared-memory system. It supports concurrent lookup, insert and remove operations. Just like the hash array mapped trie, it uses the entire 32-bit space for hash values thus having low risk of hashcode collisions. Each node may branch to up to 32 sub tries. To conserve memory, each node contains a 32 bits bitmap where each bit indicates the presence of a branch followed by an array of length equal to the Hamming weight of the bitmap.

Keys are inserted by doing an atomic compare-and-swap operation on the node which needs to be modified. To ensure that updates are done independently and in a proper order, a special indirection node (an I-node) is inserted between each regular node and its subtries.



The figure above illustrates the Ctrie insert operation. Trie A is empty - an atomic CAS instruction is used to swap the old node C1 with the new version of C1 which has the new key *k1*. If the CAS is not successful, the operation is restarted. If the CAS is successful, we obtain the trie B. This procedure is repeated when a new key *k2* is added (trie C). If two hashcodes of the keys in the Ctrie collide as is the case with *k2* and *k3*, the Ctrie must be extended with at least one more level - trie D has a new indirection node I2 with a new node C2 which holds both colliding keys. Further CAS instructions are done on the contents of the indirection nodes I1 and I2 - such CAS instructions can be done independently of each other, thus enabling concurrent updates with less contention.

The Ctrie is defined by the pointer to the root indirection node (or a root I-node). The following types of nodes are defined for the Ctrie:

```
structure INode {
  main: CNode
}

structure CNode {
  bmp: integer
  array: Branch[2^W]
}

Branch: INode | SNode

structure SNode {
  k: KeyType
  v: ValueType
}
```

A C-node is a branching node. It typically contains up to 32 branches, so *W* above is 5. Each branch may either be a key-value pair (represented with an S-node) or another I-node. To avoid wasting 32 entries in the branching array when some branches may be empty, an integer bitmap is used to denote which bits are full and which are empty. The helper method *flagpos* is used to inspect the relevant hashcode bits for a given level and extract the value of the bit in the bitmap to see if its set or not - denoting whether there is a branch at that position or not. If there is a bit, it also computes its position in the branch array. The formula used to do this is:

```
bit = bmp & (1 << ((hashcode >> level) & 0x1F))
pos = bitcount((bit - 1) & bmp)
```

Note that the operations treat only the I-nodes as mutable nodes - all other nodes are never changed after being created and added to the Ctrie.

Below is an illustration of the pseudocode of the insert operation:

```
def insert(k, v)
  r = READ(root)
  if iinsert(r, k, v, 0, null) = RESTART insert(k, v)
```

```
def iinsert(i, k, v, lev, parent)
  cn = READ(i.main)
  flag, pos = flagpos(k.hc, lev, cn.bmp)
  if cn.bmp & flag = 0 {
    ncn = cn.inserted(pos, flag, SNode(k, v))
    if CAS(i.main, cn, ncn) return OK
    else return RESTART
  }
  cn.array(pos) match {
    case sin: INode => {
      return iinsert(sin, k, v, lev + W, i)
    case sn: SNode =>
      if sn.k ≠ k {
        nsn = SNode(k, v)
        nin = INode(CNode(sn, nsn, lev + W))
        ncn = cn.updated(pos, nin)
        if CAS(i.main, cn, ncn) return OK
        else return RESTART
      } else {
        ncn = cn.updated(pos, SNode(k, v))
        if CAS(i.main, cn, ncn) return OK
        else return RESTART
      }
  }
```

The *inserted* and *updated* methods on nodes return new versions of the C-node with a value inserted or updated at the specified position, respectively. Note that the insert operation above is tail-recursive, so it can be rewritten as a while loop. Other operations are described in more detail in the original paper on Ctries.[1][5]

The data-structure has been proven to be correct[1] - Ctrie operations have been shown to have the atomicity, linearizability and lock-freedom properties. The lookup operation can be modified to guarantee wait-freedom.

Advantages of Ctries [edit]

Ctries have been shown to be comparable in performance with concurrent skip lists,[2][4] concurrent hash tables and similar data structures in terms of the lookup operation, being slightly slower than hash tables and faster than skip lists due to the lower level of indirections. However, they are far more scalable than most concurrent hash tables where the insertions are concerned.[1] Most concurrent hash tables are bad at conserving memory - when the keys are removed from the hash table, the underlying array is not shrunk. Ctries have the property that the allocated memory is always a function of only the current number of keys in the data-structure.[1]

Ctries have logarithmic complexity bounds of the basic operations, albeit with a low constant factor due to the high branching level (usually 32).

Ctries support a lock-free, linearizable, constant-time snapshot operation,[2] based on the insight obtained from persistent data structures. This is a breakthrough in concurrent data-structure design, since existing concurrent data-structures do not support snapshots. The snapshot operation allows implementing lock-free, linearizable iterator, size and clear operations - existing concurrent data-structures have implementations which either use global locks or are correct only given that there are no concurrent modifications to the data-structure. In particular, Ctries have an O(1) iterator creation operation, O(1) clear operation, O(1) duplicate operation and an amortized O(logn) size retrieval operation.

Problems with Ctries [edit]

Most concurrent data structures require dynamic memory allocation, and lock-free concurrent data structures rely on garbage collection on most platforms. The current implementation[4] of the Ctrie is written for the JVM, where garbage collection is provided by the platform itself. While it's possible to keep a concurrent memory pool for the nodes shared by all instances of Ctries in an application or use reference counting to properly deallocate nodes, the only implementation so far to deal with manual memory management of nodes used in Ctries is the common-lisp implementation cl-ctrie, which implements several stop-and-copy and mark-and-sweep garbage collection techniques for persistent, memory-mapped storage. Hazard pointers are another possible solution for a correct manual management of removed nodes. Such a technique may be viable for managed environments as well, since it could lower the pressure on the GC. A Ctrie implementation in Rust makes use of hazard pointers for this purpose.[6]

Implementations [edit]

A Ctrie implementation[4] for Scala 2.9.x is available on GitHub. It is a mutable thread-safe implementation which ensures progress and supports lock-free, linearizable, O(1) snapshots.

- A data-structure similar to Ctries has been used in ScalaSTM,[7] a software transactional memory library for the JVM.
- The Scala standard library includes a Ctries implementation since February 2012.[8]
- Haskell implementation is available as a package[9] and on GitHub.[10]
- A standalone Java implementation is available on GitHub.[11]
- CL-CTRIE is the Common Lisp implementation is available on GitHub.[12]
- An insert-only Ctrie variant has been used for tabling in Prolog programs.[13]
- Go implementation is available as a standalone package [14]
- A Rust implementation [6] uses hazard pointers in its implementation to achieve lock-free synchronization.

History [edit]

Ctries were first described in 2011 by Aleksandar Prokopec.[1] To quote the author:

Ctrie is 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.

In 2012, a revised version of the Ctrie data structure was published,[2] simplifying the data structure and introducing an optional constant-time, lock-free, atomic snapshot operation.

References [edit]

1. ^ [a] [b] [c] [d] [e] [f] Prokopec, A. et al. (2011) Cache-Aware Lock-Free Concurrent Hash Tries. Technical Report, 2011.
2. ^ [a] [b] [c] [d] [e] Prokopec, A., Bronson N., Bagwell P., Odersky M. (2011) Concurrent Tries with Efficient Non-Blocking Snapshots
3. ^ Prokopec, A. et al. (2011) Lock-Free Resizeable Concurrent Tries. The 24th International Workshop on Languages and Compilers for Parallel Computing, 2011.
4. ^ [a] [b] [c] [d] Prokopec, A. JVM implementation on GitHub
5. ^ http://axel22.github.io/resources/docs/lcpc-ctries.ppt
6. ^ [a] [b] Rust Ctrie implementation at GitHub
7. ^ N. Bronson ScalaSTM
8. ^ TrieMap.scala
9. ^ Haskell ctrie package
10. ^ GitHub repo for Haskell Ctrie
11. ^ GitHub repo for Java Ctrie
12. ^ GitHub repo for Common Lisp Ctrie
13. ^ Miguel Areias and Ricardo Rocha, A Lock-Free Hash Trie Design for Concurrent Tabled Logic Programs
14. ^ Go Ctrie package

Categories: Associative arrays | Hashing