Lock-free Radix Trees for IP Routing

Damian Ho (xho) & Jessica Ruan (jwruan)

Summary

We implemented concurrent radix trees for IP routing and compared its performance to coarse-grained locking, fine-grained locking, and lock-free variants on the PSC Bridges-2 machines. Our lock-free implementation uses the single-word compare-and-swap (CAS) instruction for synchronization.

Background

IP Routing

In IP Routing, a router has to determine the next hop for a packet based on the IP address of its destination. To do this, the router uses a routing table that maps IP addresses to routers.

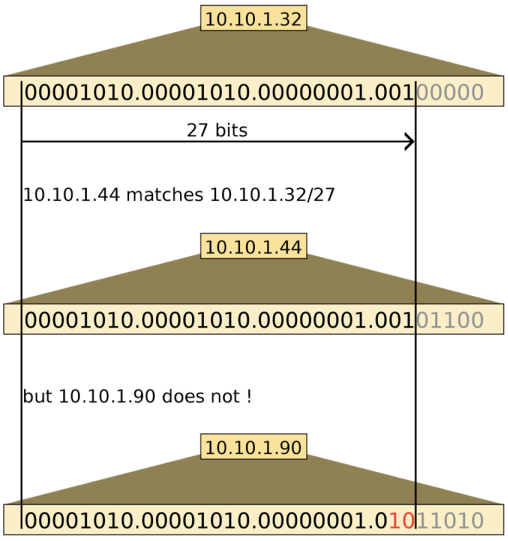


Figure 1: IP Address Matching

Routes are represented in CIDR notation, e.g. 10.10.1.32/27 where the 27 represents the number of bits in the network prefix (prefix length, a.k.a. cidr). This means that any IP address that has the same first 27 bits *matches* the route. In **Figure 1**, 10.10.1.44 matches the route 10.10.1.32/27, while 10.10.1.90 does not.

In the case that multiple routes match an IP address, the routing table returns the most specific route, or more rigorously, the route with the largest cidr value.

Thus, the routing table supports the following operations:

- `Insert(ip, cidr, router)`: associate the route `ip/cidr` with `router`
- `Delete(ip, cidr)`: remove the router associated with the route `ip/cidr`, if any
- `Lookup(ip)`: find the next-hop router for `ip`

We refer to `Insert`s and `Delete`s as "updates," or "update operations."

Radix Tree

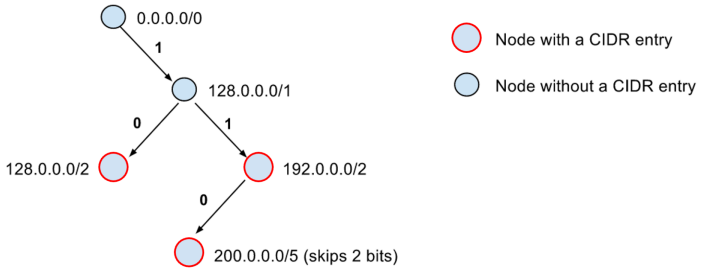


Figure 2: Structure of a Radix Tree

A commonly used data structure for the routing table is the radix tree, a specialization of a binary trie. Each node represents a route, but does not necessarily contain an entry if it serves only as a "branching point". In the figure above, the nodes corresponding to the routes 128.0.0.0/2, 192.0.0.0/2, and 200.0.0.0/5 contain entries; the rest do not. As a space optimization, any node that is an only child is merged with its parent if the parent does not have an entry.

The cidr increases with depth, which means that the best match is the deepest entry associated with a matching route. The search complexity is thus $O(k)$ where k is the number of bits in an IP address, e.g. $k = 32$ for IPv4 and $k = 128$ for IPv6.

Challenge

Update operations, comprised of inserts and deletes, can cause the structure of the radix tree to change in a non-trivial fashion. It is important to ensure that concurrent updates do not leave the tree in an inconsistent state. We also want to ensure that lookups do not view a tree in an inconsistent state.

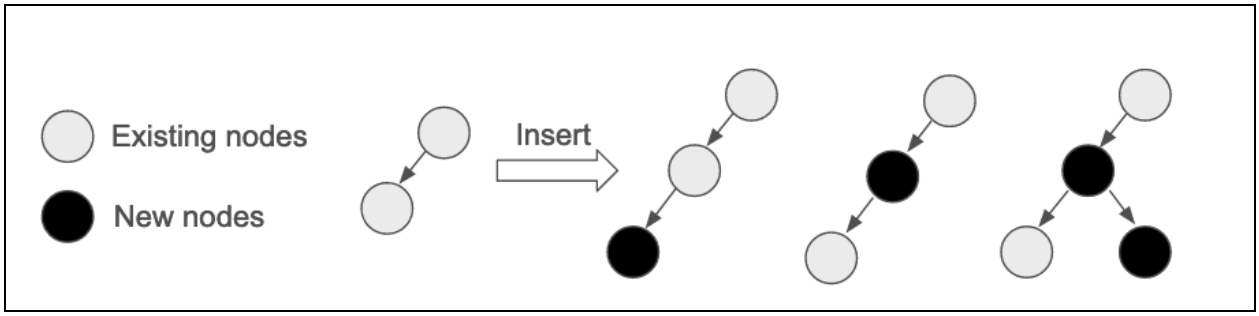


Figure 3: Possible outcomes after performing an insert operation with a new route

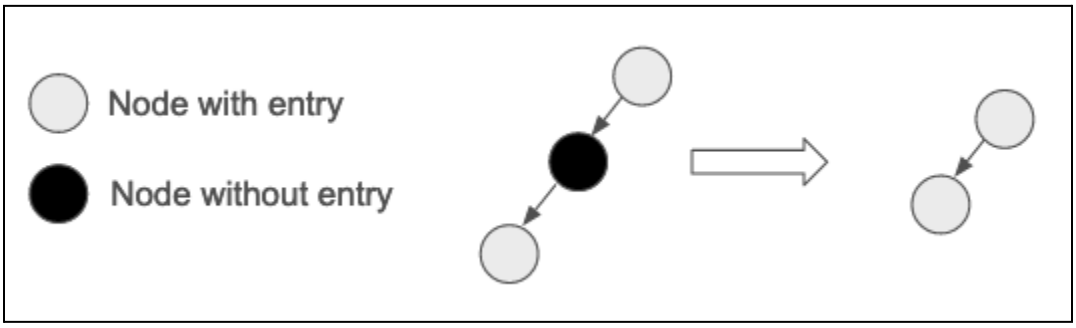


Figure 4: Parent with no entry is merged with an only child

One solution is to use synchronization primitives such as locks. With coarse-grained locking, a simple approach is to use a global read-write lock on the entire tree to ensure that multiple readers or at most one writer can access the tree concurrently. However, this severely limits parallelism especially when updates are more frequent.

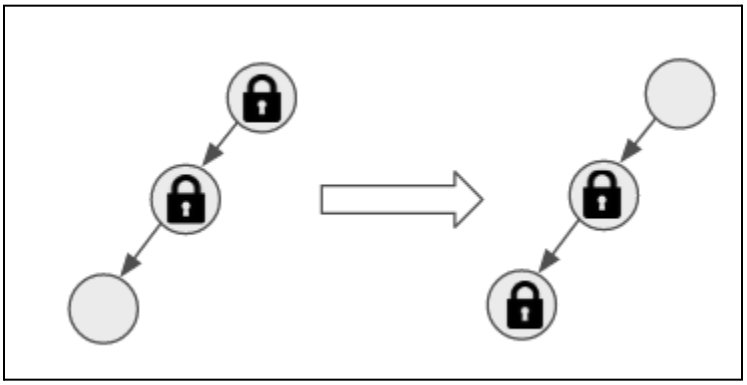


Figure 5: Hand-over-hand Locking

A better approach is to use fine-grained locking, where each node has a lock and threads only obtain locks when strictly necessary. We can perform hand-over-hand locking, where a lock is acquired on a parent and its child, and the lock on the parent is only released once it is deemed safe to do so – when we are confident that we do not need to backtrack to the parent, and that necessary updates to the parent have been made. Locks are acquired and released in this

fashion as we traverse down the tree, enabling updates to operate concurrently on different portions of the tree.

Even better would be a lock-free implementation that guarantees system-wide progress even in the presence of multiple thread failures. We can use the compare-and-swap (CAS) instruction to achieve this, using the idea of marking individual child pointers from Velamuri [1]. This facilitates system-wide progress since any thread that is blocked by an ongoing operation can help to finish the ongoing operation before restarting. Furthermore, multiple threads can update a node simultaneously, improving parallelism. This is the approach that we will modify and present in this paper.

Approach

Overview

We make use of C++23 for its `atomic` library, which supports the CAS method `compare_exchange_strong` and atomic read/write instructions. It also supports useful helper methods such as `std::byteswap`, which facilitates the conversion of IP addresses from big-endian (network byte order) to little-endian (host byte order). We targeted our code to compile on our local, GHC, and PSC Bridges-2 machines. Development and debugging were done on our local and GHC machines, while final benchmarking was performed on the PSC Bridges-2 machines.

We started by implementing a thread-unsafe radix tree by examining the source code of WireGuard [3] [4], a modern Linux VPN. We proceeded to create thread-safe variants of the radix tree through the use of coarse-grained locking and fine-grained locking, as described in the previous section.
Finally, by studying the algorithm described by Velamuri [1] and its implementation [2], we implemented a lock-free CAS radix tree. However, there are several notable differences between their implementation and ours, as ours is specifically targeted for IP routing.

	Velamuri	**Our Work**
Alphabet Size	Arbitrary alphabet size, general-purpose	Binary tree, specialized for IP routing
CAS mechanism	GCC's builtin `__sync_bool_compare_and_swap`	`std::atomic`'s `compare_exchange_strong`
Entries are stored at …	Leaf nodes	Potentially all nodes
Upon inserting a duplicate key …	Insert fails	Insert succeeds, updates the corresponding entry

Table 1: Comparison of Algorithms

For simplicity, Velamuri's algorithm assumes that memory allocated to nodes removed from the tree is not reclaimed. This means that new nodes will have unique addresses, avoiding the ABA problem. We make the same simplification in our implementation, as that is not the focus of this paper. However, we note that the read-copy-update (RCU) mechanism is used in practice to safely reclaim memory [5].

In particular, WireGuard's [3] and the Linux Kernel's [6] implementation of radix trees only use the RCU mechanism. This enables lockless lookups, but unlike our CAS implementation, locking is still necessary for updates in RCU.

Algorithm

In this section, we present the data structures used in our algorithm and the implementations of the `Insert`, `Delete`, and `Lookup` methods.

While there are similarities to Velamuri's algorithm [1], we will repeat the details for exposition, highlighting notable differences.

Data Structures

Our radix tree consists of `Node` objects, each storing the following fields: `key` of type `KEY` (in practice, an `Int` array) to store the route's IP address, `cidr` of type `Int` to store the route's cidr, `peer` of type `PeerPtr` to store the associated router, if any, and an array of 2 child pointers.

For each child pointer, we maintain two boolean values *flag* and *freeze*, which indicate the state of an ongoing remove operation. This information enables threads to assist in completing the ongoing operations of other threads. We store these two values in the lower-order bits of the child pointers, as such lower-order bits are unused due to alignment.

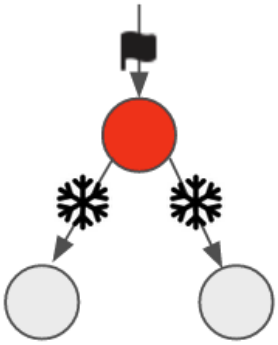


Figure 6: Removal of a node. The pointer to the node is flagged, and the node's child pointers are *frozen*.

In particular, if a node is undergoing removal, marked in red in **Figure 6**, the pointer pointing to it will have the *flag* bit set, while all of its child pointers will have the *freeze* bits set. For a child pointer, we will use the notation *<ptr, flag, freeze>* to represent the values of the raw pointer, *flag* bit, and *freeze* bit respectively. The wildcard character `*` will be used when a bit's value is indeterminate. The shorthand `<ptr, *, *>` is used when we wish to retrieve the raw pointer `ptr`.

We use a `SeekRecord` structure to record the result of a tree traversal. It records `curr`, the last node traversed, its parent `par`, and its grandparent `gPar`. It also records `currIndex` and `parIndex`, the index of `curr` in `par`'s child array and `par` in `gPar`'s child array respectively.

We also initialize a root node corresponding to the route 0.0.0.0/0 (for IPv4) which will always be the root of the tree, simplifying our implementation. We ensure that the root is never removed.

Differences from Velamuri's algorithm: Nodes additionally store `ip`, `cidr`, and `peer` fields for IP routing.

```
C/C++
struct Node {
  KEY ip;
  Int cidr;
  PeerPtr peer;
  NodePtr <ptr, flag, freeze> child[2];
};
struct SeekRecord {
  NodePtr gPar, par, curr;
  Bool parIndex, currIndex;
};
// Never removed
NodePtr root = new Node(NULL_IP, 0, NULL);
```

Algorithm 1: Structure declaration and root initialization

Search

We define a helper method `Search` which locates the node with a particular ip/cidr if such a node exists. It traverses the tree starting from the root, terminating once it finds an exact match or if it goes past the point where the node should have been. `Search` can also locate a target node.

It makes use of the node's `prefixMatches` method, which checks whether the provided key agrees with the node's `ip` on the first `cidr` bits. It also uses the node's `choose` method, which determines the next child to traverse given the provided key, based on the value of `cidr`.

If an exact match exists, `curr` will point to the target node. Otherwise, `par` points to the last node on the path to where the target node would have been. In this case, curr is either a sibling or child of the target node (possibly `NULL`).

```
C/C++
Void Search(KEY ip, Int cidr, SeekRecordPtr record) {
  NodePtr <gPar, *, *> = root, <par, *, *> = root, <curr, *, *> = root;
  Bool parIndex = false, currIndex = false;

  while (curr != NULL and curr->cidr <= cidr and curr->prefixMatches(ip)) {
    if (curr->cidr == cidr) then break; // Exact match
    gPar = par, par = curr, parIndex = currIndex;
    currIndex = node->choose(ip);
```

```
    <curr, flagged, frozen> = curr->child[currIndex];
  }
  *record = {gPar, par, <curr, flagged, frozen>, parIndex, currIndex};
}

// Alternate version of Search, finds a specific node
Void Search(NodePtr node, SeekRecordPtr record) {
  Search(node->ip, node->cidr, record);
}

Bool Contains(KEY ip, Int cidr, SeekRecordPtr record) {
  Search(ip, cidr, record);
  NodePtr <curr, *, *> = record->curr;
  return curr != NULL and curr->cidr == cidr and curr->prefixMatches(key);
}
```

Algorithm 2: `Search` and `Contains` methods

Contains

As presented in **Algorithm 2**, `Contains` runs `Search` as a subroutine to traverse the tree. If the last traversed node is not `NULL`, has the correct `cidr`, and has the same prefix as `key`, it is an exact match and it returns *true*. Otherwise, it returns *false*.

Differences from Velamuri's algorithm: The check for `Contains` is different since entries can be stored at non-leaf nodes. Thus, instead of an exact match of `curr->ip` and `key`, it suffices for only the prefix to match.

Lookup

Lookup performs a simple traversal of the radix tree to locate the deepest matching route that has an entry.

```C/C++
PeerPtr Lookup(KEY key) {
  PeerPtr found = NULL;

  NodePtr curr = root;
  while curr != NULL and curr->prefixMatches(key) do
    PeerPtr peer = curr->peer;
    if peer != NULL then
      found = peer;
    end
    if curr->cidr == NUM_BITS then // NUM_BITS = 32 for IPv4, 128 for IPv6
      break;
    end
    Bool idx = curr->choose(key);
    curr = curr->child[idx];
  end

  return found;
}
```

Algorithm 3: Lookup method

Differences from Velamuri's algorithm: This is a method unique to our implementation and does not appear in [1], as this is a method specific to IP routing.

Remove

The `Remove` method removes the entry associated with the route ip/cidr, if any. If the route is not found, the method returns *false*. Otherwise, it checks if the node is *flagged* or *frozen* and helps out with the respective removal operation if so. To do this, it makes use of the `HelpFreeze` method to freeze all children of the node being removed. The `RemoveNode` method is then called on its parent to remove the node. Finally, `Remove` restarts to find the new location of the node.

Once the entry is successfully removed, the `MakeConsistent` method is called to remove the node from the radix tree if it has less than two children. This helps to maintain the space efficiency of radix trees. It recursively calls itself on a node's parent as applicable.

```
C/C++
// Returns true if an entry was removed
Bool Remove(KEY ip, Int cidr) {
  SeekRecordPtr record = new SeekRecord();
  if (!Contains(ip, cidr, record)) then return false;
  NodePtr <gPar, *, *> = record->gPar;
  NodePtr <par, *, *> = record->par;
  NodePtr <curr, flagged, frozen> = record->curr;
  Int parIndex = record->parIndex, currIndex = record->currIndex;
  if flagged then // Help to remove node, then try again
    curr->HelpFreeze();
    par->RemoveNode(curr, currIndex);
    return Remove(ip, cidr);
  else if frozen then // Help to remove parent, then try again
    par->HelpFreeze();
    gPar->RemoveNode(par, parIndex);
    return Remove(ip, cidr);
  else
    curr->peer = NULL;
    MakeConsistent(curr, record);
    return true;
  end
}
```

Algorithm 4: `Remove` method

Differences from Velamuri's algorithm: Since entries can be stored at non-leaf nodes, we atomically set the entry to `NULL` rather than performing a CAS to remove a leaf. Furthermore, we handle the case where the node is *flagged*. This case does not arise in [1] since leaf nodes, where all entries exist, are never *flagged*.

```
C/C++
Void MakeConsistent(NodePtr child, SeekRecordPtr record) {
  Int childCount = child->CheckChild();
  if child == root or child->peer != NULL or childCount > 1 then return;

  // Locate child and its parent
  Search(child, record);
  NodePtr parent = record->par;
  Bool idx = record->currIndex;
  NodePtr <newChild, flagged, frozen> = record->curr;
  if newChild != child then return; // Child was removed

  if frozen then // Help to remove parent
    parent->HelpFreeze();
    Search(parent, record);
    NodePtr <newParent, *, *> = record->curr;
    if newParent == parent then
      NodePtr <gPar, *, *> = record->par;
      Bool parIndex = record->currIndex;
      gPar->RemoveNode(parent, parIndex);
      MakeConsistent(parent, record); // Recursively check
    end
    return MakeConsistent(child, record); // Parent removed, try again
  // Flag child to remove it
  else if flagged or parent->CAS(idx, <child, 0, 0>, <child, 1, 0>) then
    child->HelpFreeze();
    parent->RemoveChild(child, idx);
    return MakeConsistent(parent, record); // Recursively check
  else if (<newChild, *, *> = parent->child[idx]) != child then
    return MakeConsistent(child, record); // Parent changed, try again
  end
  // Otherwise, another thread just flagged it. We don't have to do anything
since other threads will help to finish the operation if necessary
}
```

Algorithm 5: `MakeConsistent` method

Differences from Velamuri's algorithm: Specialization of methods for the binary alphabet, and `MakeConsistent` now always locates the node to retrieve necessary parent information.

```
C/C++
Void Node::HelpFreeze() {
  Int idx = 0;
  while idx < 2 do
    NodePtr <curr, flagged, frozen> = child[idx];
    if flagged then // Help to remove child
      curr->HelpFreeze();
      RemoveNode(curr, idx);
    else if frozen or CAS(idx, <curr, 0, 0>, <curr, 0, 1>) then
      idx++;
    end
  end
}

Void Node::RemoveNode(NodePtr node, Bool idx) {
  Int childCount = node->CheckChild();
  NodePtr ptr = NULL;
  Bool copyPeer = false;

  if childCount == 1 then
    ptr = node->child[0] == NULL ? node->child[1] : node->child[0];
  else if childCount == 2 then
    ptr = new Node(node->ip, node->cidr, node->peer);
    Int idx = 0;
    while idx < 2 do
      NodePtr <childPtr, *, *> = node->child[idx];
      ptr->child[idx] = childPtr; // Unfreeze children
      idx++;
    end
    copyPeer = true;
  end

  if CAS(idx, <node, 1, 0>, <node, 0, 0>) then
    if copyPeer then // In case node's peer was updated in the meanwhile
      ptr->peer.CAS(NULL, node->peer);
      // If CAS fails, then we already have a new peer; do nothing
    end
  end
}

Int Node::CheckChild() {
  return (<child0, *, *> != NULL) + (<child1, *, *> != NULL);
}
```

Algorithm 6: `HelpFreeze`, `RemoveNode`, and `CheckChild` class methods

Observe that `RemoveNode` is only called after freezing child pointers with `HelpFreeze`, to avoid race conditions where new children might be inserted at the same time as `CheckChild` is counting the number of children. If it turns out that a node has two children after freezing its children, we unfreeze it by creating a new node to replace it. `HelpFreeze` also checks if any children are marked for removal, and if so, assists with the removal operation before proceeding.

Insert

Similarly to the `Remove` method, `Insert` assists ongoing removal operations. There are then three possible cases for insertion. The route being inserted may either be a child, parent, or sibling of the curr node found by `Contains`. In either case, we construct the necessary nodes and perform a CAS to insert them. In case of contention, `Insert` restarts to determine the new point of insertion.

```C/C++
// Returns true if a new route was inserted
Bool Insert(KEY key, Int cidr, PeerPtr peer) {
  SeekRecordPtr record = new SeekRecord();
  Bool exists = Contains(key, cidr, record);
  NodePtr <gPar, *, *> = record->gPar;
  NodePtr <par, *, *> = record->par;
  NodePtr <curr, flagged, frozen> = record->curr;
  Bool parIndex = record->parIndex, currIndex = record->currIndex;
  if flagged then // Help to remove node, then try again
    curr->HelpFreeze();
    par->RemoveNode(curr, currIndex);
    return Insert(key, cidr, peer);
  else if frozen then // Help to remove parent, then try again
    par->HelpFreeze();
    gPar->RemoveNode(par, parIndex);
    return Insert(key, cidr, peer);
  else if exists then
    curr->peer = peer; // Update existing
    return false;
  end

  NodePtr ptr;
  NodePtr newNode = new Node(key, cidr, peer);
  if curr == NULL then // Case 1: par -> newNode
    ptr = newNode;
  else
    Int common = CommonBits(curr->ip, key); // Longest matching prefix
    cidr = min(cidr, common);
    if newNode->cidr == cidr then // Case 2a: par -> newNode -> curr
      Bool idx = newNode->choose(curr->ip);
      newNode[idx] = curr;
      ptr = newNode;
    else // Case 2b: par -> newParent -> curr + newNode
      NodePtr newParent = new Node(key, cidr, NULL);
      Bool bit = newParent->choose(key);
      newParent->child[bit] = newNode;
```

```
      newParent->child[bit ^ 1] = curr;
      ptr = newParent;
    end
  end

  // Insert node
  if par->CAS(currIndex, <curr, 0, 0>, <ptr, 0, 0>) then
    return true;
  end

  // Contention, retry
  return Insert(key, cidr, peer);
}
```

Algorithm 7: `Insert` method

Differences from Velamuri's algorithm: The node creation logic has more cases (i.e. case 2a) since we might be inserting internal nodes. Furthermore, we support inserting an existing route – this updates its entry.

Correctness

Node Removal

The crucial invariant is that before a node can be removed, its pointer stored in its parent must be *flagged* and all of its child pointers must be *frozen*.

The *flag/freeze* mechanism ensures several things:

1. *The parent will not be removed*. For the parent to be removed, it has to *freeze* its children. At that point, `HelpFreeze` would notice that the node is *flagged* and complete the removal operation before proceeding.
2. *The parent will not change*. `Insert` cannot change the parent of a node–in other words, make the parent point to something else–if the node is *flagged*.
3. *No child will be inserted*. An entry in the node's child array cannot be updated during `Insert` if the *freeze* bit is set.
4. *No child will be removed*. A child node cannot be *flagged* for deletion during `MakeConsistent` if it has the *freeze* bit set.

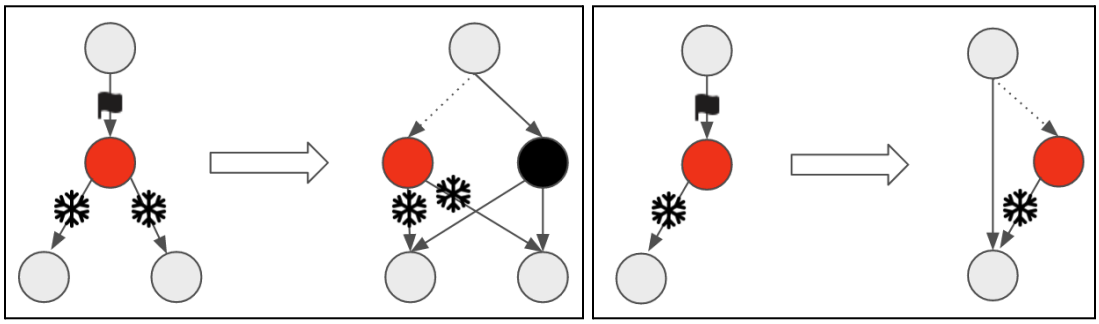


Figure 7 (Left): two non-null children due to a race; make a copy and unset the freeze bits
Figure 8 (Right): one non-null child; set parent pointer to that child.

In summary, the parent and children of a node remain stable when a node is marked for removal. Then, a recount of non-null children is performed to determine whether to follow through with the removal (**Figures 7 & 8**). In either case, the *flag* bit of the (copy of the) node and the *freeze* bit of its children are simultaneously cleared.

Sequential Correctness

To test the correctness of the lock-free radix tree (and other variants) when run sequentially, we created tests adapted from those used by WireGuard's Go implementation [4]. We also wrote stress tests that compared the output of different radix tree variants, including the naïve linked list and binary trie implementations.

Concurrent Correctness

After each concurrent test, we run validation checks to check the structural integrity of the radix tree. For example, cidr values should increase with depth, and a node's children should be at the expected index in its child array. The expected size of the radix tree was also tracked based on the insertions and deletions that were completed, and we compared it against the final size of the radix tree.

Non-blocking Progress

A proof sketch is provided by Velamuri [1] for their algorithm. Given the similarity of our algorithm, the same argument broadly applies.

Results

Simulated Workload

Workload Configuration:

- t: number of threads
- N: number of operations per thread
- $r1$: ratio of update to lookup operations, with an update operation being an `Insert` or a `Delete`
- $r2$: ratio of `Insert` to `Delete`

A total of t test files are generated, one per thread, where each file contains N operations, one per line. Within each file

- Each operation is an update (Insert/Delete) with probability $r1$ and a lookup otherwise. If the operation is an update, it is an `Insert` with probability $r2$ and a `Delete` otherwise.
- The keys to be inserted or deleted are drawn from a universe of 14 keys. Upon insertion, keys are uniformly randomly selected from the set of uninserted keys, while upon deletion or lookup, keys are uniformly randomly selected from the set of inserted keys.
- When the size of the trie crosses a threshold, the ratio of insertions to deletions, $r2$, is inverted. That is, if over 75% of the keys in the universe have been inserted and rate of insertions to deletions is greater than 0.5, $r2 := 1 - r2$. Similarly, if over 75% of the keys in the universe have been uninserted and the rate of insertions to deletions is lower than 0.5, $r2 := 1 - r2$.

Next, t threads are created, with each thread executing f files. Each thread loads the contents of the files into memory before executing the operations on the specified radix tree. To account for the disk I/O overhead, timing begins after the thread has loaded the files into memory.

Metrics measured:

- **Throughput:** We calculate throughput as $\frac{total\ number\ of\ operations}{overall\ duration}$, with the total number of operations (across all threads) being $t \cdot N$, and overall duration being the maximum duration taken for any thread to finish.
- **Single-threaded performance:** As a comparison, we run the single-threaded radix tree implementation sequentially on all t files to get the overall sequential duration.
- **Average time per operation:** We measure the average duration for the insert, update, and delete operations of each radix trie implementation.

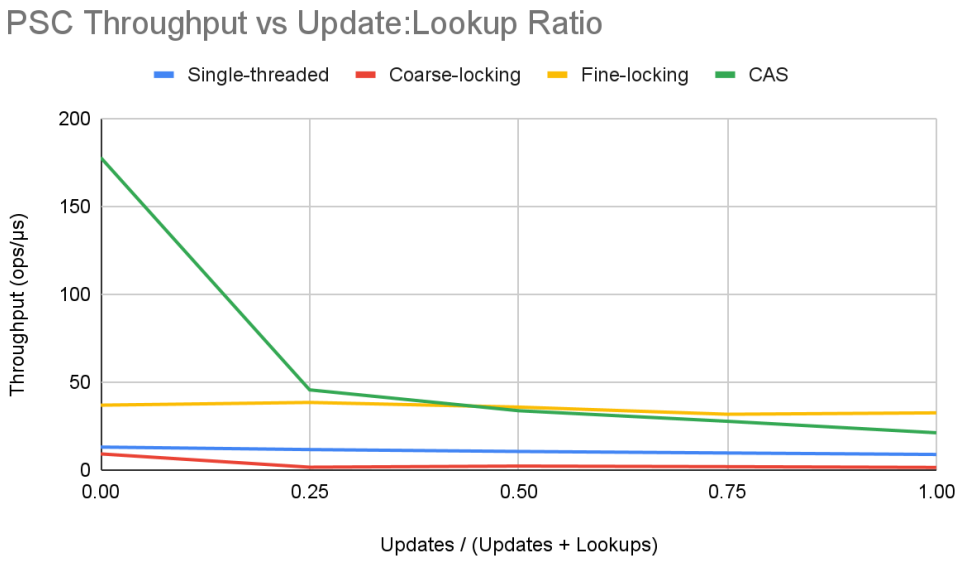
Experiments

The experiments were performed on a PSC Bridges-2 Regular Memory machine. The machine was equipped with RHEL 8.8 OS, 256 GB RAM, and two AMD EPYC 7742 CPUs (2.25 GHz) of 64 cores each, for a total of 128 cores per node. Hyperthreading is enabled, yielding 256 logical cores in total. Each CPU has a 32KB L1d and L1i cache, with a 16.4MB LLC.

All our implementations are written in C++23 and compiled using g++ 8.5.0 with optimization level O3. Random numbers were generated (for IP/cidr values) using the Mersenne Twister 19937 generator with 32-bit integers.

Experiment 1: Throughput vs Update:Lookup Ratio

For this experiment, we fix $t = 8$, $N = 50000$, and $r2 = 0.75$.
We measure throughput as we vary the update-to-lookup ratio $r1 \in \{0, 0.25, 0.5, 0.75, 1.00\}$.

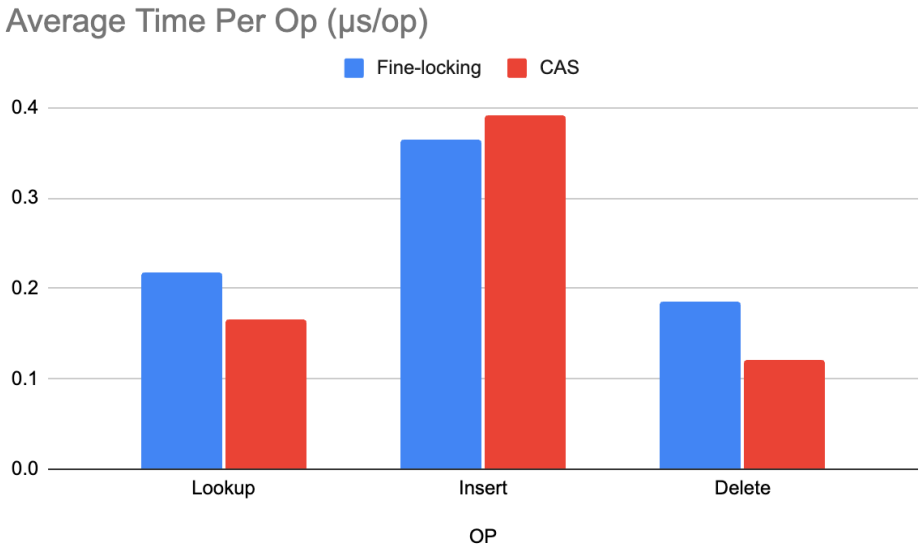


As the update-to-lookup ratio increases, throughput generally decreases for all implementations, as updates are expensive relative to lookups given that they modify the tree structure.

We see that the CAS implementation has the highest throughput, followed by the fine-grained locking implementation. Coarse-grained locking consistently has the lowest throughput since only one update can occur at a time, which severely restricts concurrency.

For lookup-heavy workloads, the CAS implementation outperforms the fine-grained locking implementation. However, as the workload becomes more update-heavy, the fine-grained locking implementation surpasses the performance of the CAS implementation.

To investigate the reason behind the difference in performances for lookup-heavy vs update-heavy workloads, we timed the average time per operation for the fine-grained locking and CAS implementations.



For lookup operations, CAS outperforms the fine-grained locking implementation. Thus, it makes sense that it performs better for lookup-heavy workloads. We speculate that this is because the fine-grained locking implementation has to acquire multiple read locks for each lookup operation, whereas the CAS implementation does not need to use any synchronization mechanism.

For update operations, insertions are the most expensive operation for both the CAS and fine-grained locking implementations. This is likely due to the overhead of allocating memory for new nodes that are created.

We note that the CAS implementation is slower for insertions. We speculate that this is because the bookkeeping performed by CAS has a greater overhead than locks.

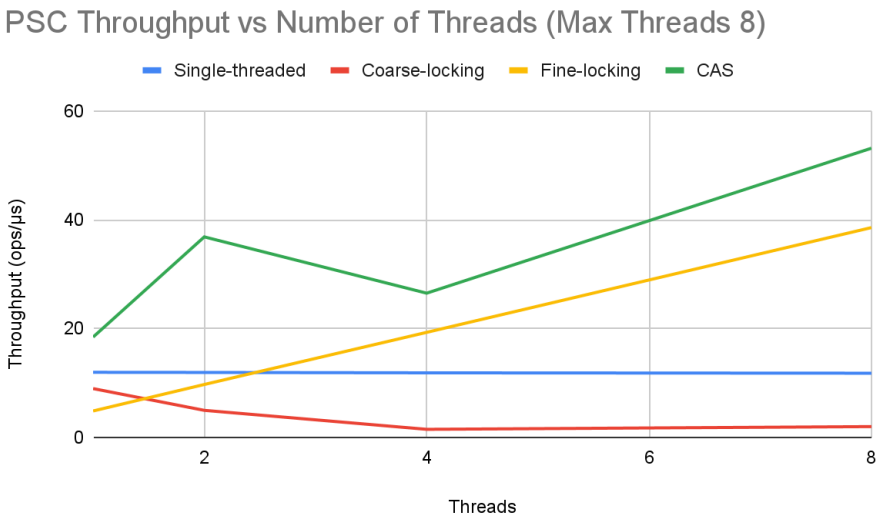
However, the CAS implementation is faster for deletions. We speculate that this is due to the fine-grained locking implementation's use of shared pointers for automatic memory reclamation, whereas the CAS implementation does not reclaim memory for simplicity.

Hence, for update-heavy workloads, the performance of the CAS implementation to the fine-grained locking implementation is dependent on the ratio of insertions to deletions.

That said, we note that lookup-heavy workloads are likely to be more common in IP-routing. Hence, the CAS implementation is a strong candidate for IP-routing due to its advantage in lookup-heavy workloads.

Experiment 2: Throughput Over Number of Threads

We fix $N = 50000$, $r1 = 0.25$, $r2 = 0.75$ and measure throughput as we vary the number of threads $t \in \{1, 2, 4, 8\}$.



As the number of threads increases, we observe that throughput generally increases since more threads are doing work at once. We also observe that the CAS and fine-grained locking implementations consistently enjoy the highest throughput due to their support for concurrent lookups and updates.

Since this workload is lookup-heavy, CAS once again outperforms the fine-grained locking implementation due to its lower overhead for lookup operations.

Conclusion

Overall, a CAS-based lock-free radix tree is viable for IP routing. It consistently has one of the highest throughputs and handles workloads of various distributions well. In addition, it scales as the number of threads increases. Although it is unlikely for many threads to be used for IP routing in practice, this is still a potential point of consideration.

The best alternative is the fine-grained locking radix tree, which has comparable but slightly lower performance in the throughput benchmarks. The lock-free radix tree was seen to have a significant advantage over the fine-grained locking implementation for lookup-heavy workloads, although the advantage diminishes as updates become more frequent.

Furthermore, the lock-free radix tree does not reclaim memory, which could be a concern for long-running applications. A possible solution is to utilize the read-copy-update (RCU) mechanism to efficiently reclaim memory that is no longer accessed by any readers.

References

1. Velamuri, V.S. (2017). Efficient Non-blocking Radix Trees. *European Conference on Parallel Processing*.
2. Velamuri, V. (2017). Source code for efficient non-blocking radix trees. https://github.com/varun1312/RadixTrees
3. Donenfeld, J. (2019). WireGuard for Linux 3.10 - 5.5. https://github.com/WireGuard/wireguard-linux-compat
4. Donenfeld, J. (2018). Go implementation of WireGuard. https://github.com/WireGuard/wireguard-go
5. McKenney, P. (2007, December 17). *What is RCU, fundamentally?*. Linux Weekly News. https://lwn.net/Articles/262464/
6. Corbet, J. (2006, March 13). *Trees I: Radix trees*. Linux Weekly News. https://lwn.net/Articles/175432/

Distribution of total credit

Damian Ho

- Design radix tree interface
- Implement thread-unsafe linked list, binary trie, radix tree
- Implement coarse-grained locking, fine-grained locking, lock-free (CAS) radix trees
- Write basic correctness tests, stress tests

Jessica Ruan

- Implement thread-unsafe radix tree
- Implement lock-free (CAS) radix tree
- Write performance benchmarks, workload generator
- Gather and plot results

Credit distribution: 50% (Damian Ho), 50% (Jessica Ruan)