Lock-free Radix Trees for IP Routing

Damian Ho (xho) & Jessica Ruan (jwruan)

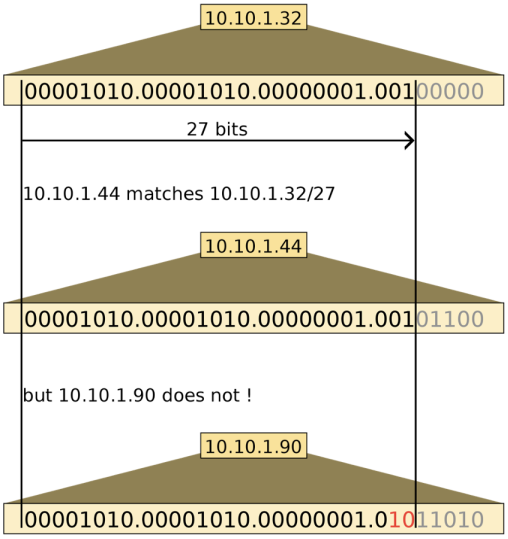
Project Web Page: https://damianhxy.github.io/15418/

Summary

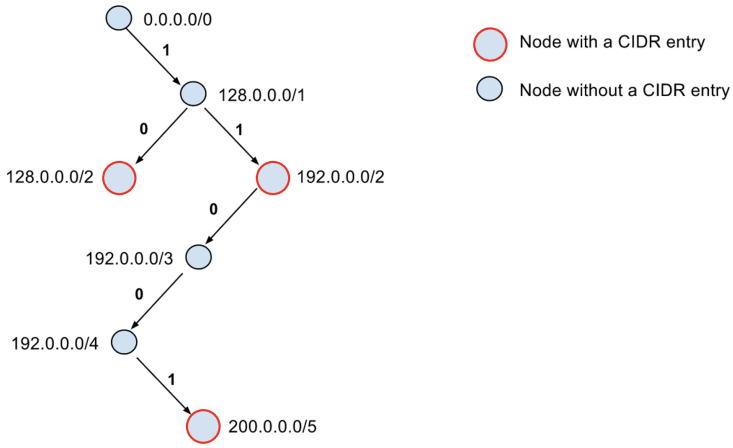
We will implement concurrent coarse-grained, fine-grained, and lock-free radix trees for IP routing. We will then compare their performance under simulated network workloads.

Background

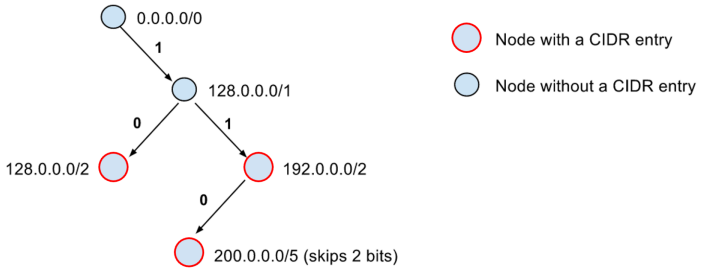
In IP Routing, the routing table is responsible for selecting the next-hop router for a datagram. It does so by choosing a matching route with the greatest subnet mask. IPv4 addresses consist of 32 bits and match a route if they share the same network identifier bits.



For example, 10.10.1.44 matches 10.10.1.32/27, since the first 27 bits are equal.



One option is to use a binary trie, which means that the best match is the deepest entry encountered. Operations thus run in time linear in the number of bits.



However, we can do better by using radix trees, which perform path compression. This means that nodes which are only childs are merged with their parents. Less time is needed to traverse the tree, speeding up operations. In the radix tree above, 200.0.0.0/5 is the left child of 192.0.0.0/2, after skipping two redundant bits.

Challenge

Other than routing table lookups for incoming packets, there can also be updates in the form of inserting or deleting routing table entries. We will start by implementing coarse-grained and fine-grained locking variants of radix trees. However, this will lead to a performance and storage overhead which we hope to alleviate through the implementation of a lock-free radix tree.

Workload

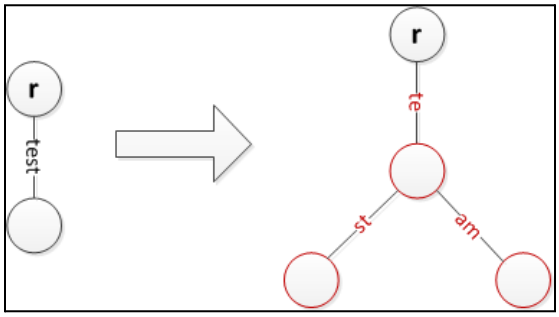
- **Communication**: We will operate on a shared memory model, thus there will be no explicit communication between threads other than to access shared variables
- **Dependencies**: Dependencies exist between threads that operate on the same subtree of the radix tree when at least one operation is an update.
- **Memory accesses**: Lookups and updates access the tree nodes in a simple path from the root node to some internal / leaf node.

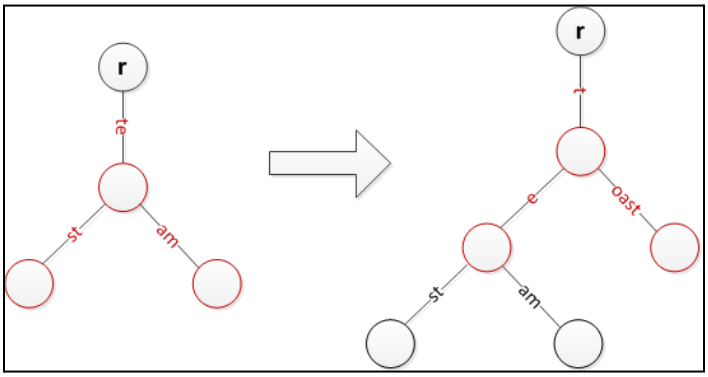
- **Locality**: Locality is greatest near the root, and decreases as depth increases.
- **Divergent execution**: In general, each lookup or update is independent of one another, thus there is little opportunity for parallelizing operations.

Constraints

We need to find a suitable locking or non-blocking schema for radix trees. Without such a schema, it might be the case that a reader performs a lookup at the same time a writer splits a node, resulting in a failed lookup.

Upon insertion, there is a possibility that the prefix of the node gets split or previous strings are moved to a lower level. Likewise upon deletion, there is a possibility that the prefix of a node gets merged with that of its parent. If a reader is accessing the same node when one of these write operations is taking place, it can result in a failed lookup.





For the fine-grained locking implementation, hand-over-hand locking is a potential schema. As implied by the name, two locks are held at a time in a hand-over-hand fashion, as though traversing monkey bars. A lock is acquired on the parent and its child, and the lock on the parent is not released until “safe” to do so. Locks are acquired and released in this fashion as we traverse down the tree.

To ensure correctness in the lock-free implementation,

- Upon insertion, proper memory ordering instructions must be issued to make sure a concurrent lookup will not find uninitialized data in the new node.
- Upon deletion, the node must remain valid and allocated for as long as it is possible that a reader may still hold a reference to the old pointer.

For all operations in the lock-free implementation, lookups must take care of two cases: one in which the lookup finds the version before the write and one in which the lookup finds the version after the write. More relaxed semantics are needed for handling these cases.

Resources

We will start by examining a Go implementation of a sequential radix tree used by WireGuard (source), an open source VPN. We will rewrite this from scratch in modern C++, to facilitate the use of locks and other synchronization primitives when implementing concurrent radix trees.

To implement the lock-free radix tree, we will examine the following papers

- Lockless radix-tree
- Velamuri, V. (2017). Efficient Non-blocking Radix Trees. In: Rivera, F., Pena, T., Cabaleiro, J. (eds) Euro-Par 2017: Parallel Processing. Euro-Par 2017. Lecture Notes in Computer Science(), vol 10417. Springer, Cham. https://doi.org/10.1007/978-3-319-64203-1_41

We might also examine the implementation by WireGuard's Linux Kernel Module (source), however it is not directly applicable as it is written in C and uses the kernel read-copy-update (RCU) synchronization mechanism. Jemalloc, a concurrent malloc implementation also uses a lock-free radix tree (source), however it is specifically tailored for its purposes.

We will test our code on both GHC and PSC Bridges in order to evaluate the performance of the implementations under varying levels of load and contention.

Goals and Deliverables

Plan to achieve

- Sequential implementation of radix tree
- Coarse-grained locking implementation of radix tree
- Fine-grained locking implementation of radix tree
- Lock-free implementation (CAS) of radix tree
- Utilities to generate workloads and test performance
- Test harness to check correctness of implementations
- Detailed comparison of radix tree performances under different workloads

Hope to achieve

- Userspace RCU lock-free implementation of radix tree, and comparison to CAS implementation
- Radix trees with radix > 2, as this will enable general use-cases

Demo: We will display graphs that demonstrate the speedup of the lock-free radix tree over the other implementations under different conditions.

The resultant data structure will be capable of concurrent lookups and updates while preserving integrity and correctness. Performance should be significantly improved under update-heavy workloads.

Platform Choice

Computer

- GHC servers for development and debugging
- PSC Bridges for benchmarking heavy workloads

Language

- C++23 for atomic operations and lock primitives

Schedule

Week	**Damian**	**Jessica**
Nov 6	Work on proposal Design radix tree interface Port linked list / binary radix tree implementation Add basic correctness tests	Work on proposal Port radix tree implementation
Nov 13 Project Proposal Due (Nov 15)	Finish proposal Create proposal website Write correctness checker Implement coarse-grained radix tree	Write performance benchmarks Write workload generator
Nov 20	Read lock-free papers Implement lock-free radix tree	Read lock-free papers Implement fine-grained locking radix tree Implement lock-free radix tree
Nov 27 Milestone Report Due (Dec 2)	Benchmark implementations Work on milestone report General purpose lock-free radix tree	Benchmark implementations Work on milestone report General purpose lock-free radix tree
Dec 4	RCU lock-free radix tree	RCU lock-free radix tree

Dec 11 Final Report Due (Dec 14) Poster Session (Dec 15)	Work on final report and poster	Work on final report and poster