Lock-free Radix Trees for IP Routing

Damian Ho (xho) & Jessica Ruan (jwruan)

Work completed

As a baseline, we wrote thread-unsafe linked list, binary trie, and radix tree implementations for the purposes of IP routing. Thereafter, we built upon the radix tree implementation to create thread-safe variants through the use of coarse-grained locking (global mutex), fine-grained locking (per-node mutex), and CAS (lock-free).

We have also written several utilities to evaluate the implementations, such as a (stress) test harness to evaluate correctness, and benchmarks to measure performance under varying levels of contention.

Goals

We are on track to producing all our deliverables, having successfully implemented two separate CAS-based lock-free radix trees. What remains is for us to scrutinize the correctness of the implementations, and compare their performances to the lock-based radix trees.

For the “nice to haves”, we no longer plan to implement a general radix tree with radix > 2, as we found an existing implementation that serves this purpose.
We still hope to implement a RCU-based lock-free radix tree implementation, which is more common in practice and simpler to implement. This would provide us with yet another point of comparison.

Plan to achieve

- Coarse-grained radix-tree implementation
- Fine-grained radix-tree implementation
- Lock-free (CAS) radix-tree implementation
- Test harness to evaluate correctness of implementations
- Performance comparison of the radix tree implementations

Hope to achieve

- Lock-free (RCU) radix-tree implementation

Poster Session

We plan to display graphs that compare the performance (e.g. speed) of lock-free radix trees to the other lock-based radix-trees under different workloads and varying levels of contention.

Preliminary Results

Simple Tests
Assign all 16 threads to the same input file (list of operations).

- **Build-IPV4**: one thread inserts 14 keys into the tree
- **Lookup-10000-IPV4**: all 16 threads perform 10000 different lookups to the tree built in Build-IPV4
- **Lookup-50000-IPV4**: all 16 threads perform 50000 different lookups to the tree built in Build-IPV4
- **Lookup-200000-IPV4**: all 16 threads perform 200000 different lookups to the tree built in Build-IPV4
- **Interleave-4-IPV4**: all 16 threads perform the following
    - Insert 14 keys into the tree. Each insertion is followed by 4 lookups.
    - Out of the 14 newly inserted keys, delete each key from the tree one by one. Each deletion is followed by 4 lookups.

	Coarse	**Fine**	**CAS 2**	**CAS 1**
Build (1 thread)	5µs	13µs	0µs	0µs
Lookup-10000-IPV4	85144µs	4µs	0µs	6µs
Lookup-50000-IPV4	442841µs	659µs	20µs	129µs
Lookup-200000-IPV4	1750917µs	647µs	1444µs	1357µs
Interleave-4-IPV4	17473µs	7µs	0µs	0µs

The time measured is the average running time of the threads.
As expected, the coarse-grained locking implementation is much slower than the other implementations due to lock contention.

It appears that fine-grained locking performs better than the lock-free implementations when handling read-heavy workloads (Lookup-10000-IPV4, Lookup-50000-IPV4, Lookup-20000-IPV4). However, for the workload that interleaves reads with writes, the lock-free implementation performs better.

Mixed Tests
Divide the 16 threads among different input files (insert, lookup, remove), and run them at the same time.

- **Mixed 1**
    - Insert-IPV4 with 25% of threads
    - Lookup-10000-IPV4 with 50% of threads
    - Remove-IPV4 with 25% of threads
- **Mixed 2**
    - Insert-IPV4 with 25% of threads
    - Lookup-50000-IPV4 with 50% of threads
    - Remove-IPV4 with 25% of threads
- **Mixed 3**
    - Insert-IPV4 with 25% of threads
    - Lookup-200000-IPV4 with 50% of threads
    - Remove-IPV4 with 25% of threads

	Coarse	**Fine**	**CAS 2**	**CAS 1**
Mixed 1	3831µs	0µs	3µs	4µs
Mixed 2	20567µs	2µs	3µs	3µs
Mixed 3	75912µs	11µs	8µs	14µs

In these scenarios, the fine-grained locking and lock-free implementations appear to have comparable performance.

Issues

We were only able to find one paper[1] that detailed the implementation of radix trees using the CAS instruction as other implementations were overwhelmingly RCU-based. The paper assumes that memory allocated to nodes is never recovered, allowing the ABA problem to be ignored. Although there exist potential approaches[2], we have adopted a similar approach of not reclaiming memory for simplicity.

Instead, we will focus our efforts on implementing a RCU-based lock-free radix tree, which would greatly simplify memory management.

---

[1] 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

[2] Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way. In Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC '15). Association for Computing Machinery, New York, NY, USA, 261–270. https://doi.org/10.1145/2767386.2767436