Project Proposal

Authors: Yuchen Liang (yuchenl3) and Hang Shu (hangshu)

URL to project homepage: https://yliang412.github.io/cart

Proposal PDF: https://yliang412.github.io/cart/files/proposal.pdf

Summary

We are aiming at implementing an efficient concurrent Adaptive Radix Tree (ART) in Rust. ART is one of the most performant main-memory index data structure on modern architecture that beats read-only search trees on lookup performance while at the same time support insertion and deletion. We are interested in exploring and implementing one of the two synchronization techniques described by the authors, namely *optimistic lock coupling* and *read-optimized write exclusion*.

Background

On a parallel architecture, it is critical to has an efficient synchronization approach so that the performance of the data structure scales with the number of processors. It is obvious that holding a mutex over the entire data structure is not the solution as it would serialize all operations and greatly limit throughput of the system.

For a tree-like data structure, finer-grained locking is a traditional synchronization approach that is relatively easy to implement. For instance, on a B+ Tree, one would have a reader-writer lock for each of the tree node locking the entire subtree. However, this approach does not scale well due to the overhead of acquiring and releasing locks on modern architecture. Lock-free data structures (e.g. skiplist), on the other hand, uses atomic operations to allow multiple threads to access and modify the underlying content without waiting for a lock. Compare to finer-grain locking, designing a completely lock-free data structure is more difficult and needs additional indirections.

We are interested in exploring the two synchronization approaches that balance between scalablity and ease of use. The first approach is optimistic lock coupling. Instead of directly taking a lock in the traversal, one would optimistically assume there are no concurrent modification and detect conflict by version counters. The operation would restart if a conflict is

detected, but in the common case, this approach can improve performance by avoiding unnecessary cache misses due to locking.

The second and more interesting approach is read-optimized write exclusion. This approach gurantees reads will never block or restart. In this scheme, a writer takes a lock to block other writers while a reader never acquire any locks. Reader-writer exclusion is instead provided by using atomic operations to access and modify internal data structures.

In addition to exploring these synchronization approaches, we are also interested in providing an efficient ART implementation for the Rust ecosystem while surveying the current language and library support for designing concurrent data structures.

Challenges

The most challenging part of making ART concurrent is that we want to the performance to scale while supporting both read and write operations at the same time. We will make extensive use of synchronization primitives and atomics to gurantee the integrity of the data structure while maintain high performance.

Concurrent data structure debugging and testing

Handling and debugging data races and deadlocks can be very tricky in a concurrent setting. It is even more difficult to verify the correctness of the data structure. We think Rust's ownership system might help us with this, but will still be quite challenging considering we will likely touch unsafe Rust.

Garbage Collection

In a concurrent setting, the memory of a deleted node cannot be immediately reclaimed because other readers might still be active. Therefore, we want to defer memory reclaimation until it is safe to do so. Unlike Java and other GC-based languages, where the language runtime takes care of the memory reclamation, garbage collection in Rust need to be handled by the programmers. Luckily, there are libraries in Rust that implements a epoch-based or simliar reclaimation scheme that could help us with this task.

Resources

We will base our implementation heavily on the original paper[1] describing the data structure and the follow-up paper[2] parallelizing the data structure. We are likely to implement garbage collection with the help of the crossbeam-epoch or the seize crate.

[1] V. Leis, et al., The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases, in ICDE, 2013. https://db.in.tum.de/~leis/papers/ART.pdf

[2] V. Leis, et al., The ART of Practical Synchronization, in DaMoN, 2016. https://db.in.tum.de/~leis/papers/artsync.pdf

Goals and Deliverables

75% Goals

- **Related works reading.** Understand the ART data structure and the synchronization approaches described in the papers.
- **Sequential implementation of ART.** Implement a version of ART with no synchronization. This version will help us understand the data structure better while serving as a reference to compare against.
- **Concurrent implementation of ART using lock coupling (LC).** As shown in the paper, the code structure for lock coupling and optimistic lock coupling are very similar. The LC implementation of ART will also serve as an alternative synchronization approach to compare against with.
- Extensive testing on implemented sequential and concurrent ART implementations.

100% Goals

75% Goals, plus:

- **Scalability analysis.** We will run microbenchmarks to investigate the scaliability of each finished implementations under low contention.
- **Contention analysis.** We will run microbenchmarks to measure the performance of each finished implementations when there are simulantaneous read and write operations.
- **Concurrent implementation of ART using optimistic lock coupling (OLC).** Modify the LC version to do OLC.
- **Benchmark against other concurrent data structures in Rust.** Our goal is to have comparable performance to both unordered and ordered concurrent data structures

(dashmap, crossbeam-skiplist).

125% Goals

100% Goals, plus:

- **Concurrent implementation of ART using read-optimized write exclusion (ROWEX).** The ROWEX synchronization approach should further improve read performance since a read will not wait or restart in this scheme.
- **Run YCSB benchmark.** In addtion to the microbenchmarks, it might be beneficial to evaluate the performance on a synthetic workload. YCSB is common set of workloads for evaluating the performance of different key-value stores, which fits our scope. We might need to adapt this benchmark into Rust since the workload generators and clients are written in Java.
- **SIMD optimizations.** SIMD-based parallel comparison can be used to further improve lookup performance.

Demo

A test workload will be ran on the sequential and parallel versions of our ART implementation, and we will specifically be looking at the speedup of the parallel versions for an increasing thread count. Our poster will also show speedup graphs on our ART implementation.

Analysis

We will be comparing speedup of our parallel implementation to the our sequential implementation. We are also interested in seeing the performance difference between tradational lock coupling, and optimistic lock coupling. If time permits, we will be adding in read-optimized write exclusion for comparison between the parallel implementations.

Platform Choice

We will do development locally on our own laptop but will use the GHC and PSC machines for testing concurrency and doing benchmark experiments on performance with a higher number of cores.

Schedule

Week	Work Item	Status
03/25 - 03/31	Related work reading	
04/01 - 04/07	Sequential implementation	
04/08 - 04/14	LC implementation + Microbenchmark setup	
04/15 - 04/21	OLC implementation	
04/22 - 04/28	Evaluation + Optimizations	
04/29 - 05/05	Benchmarking + Final Report	
05/06	Poster Session	