# Carnegie Mellon University
School of Computer Science

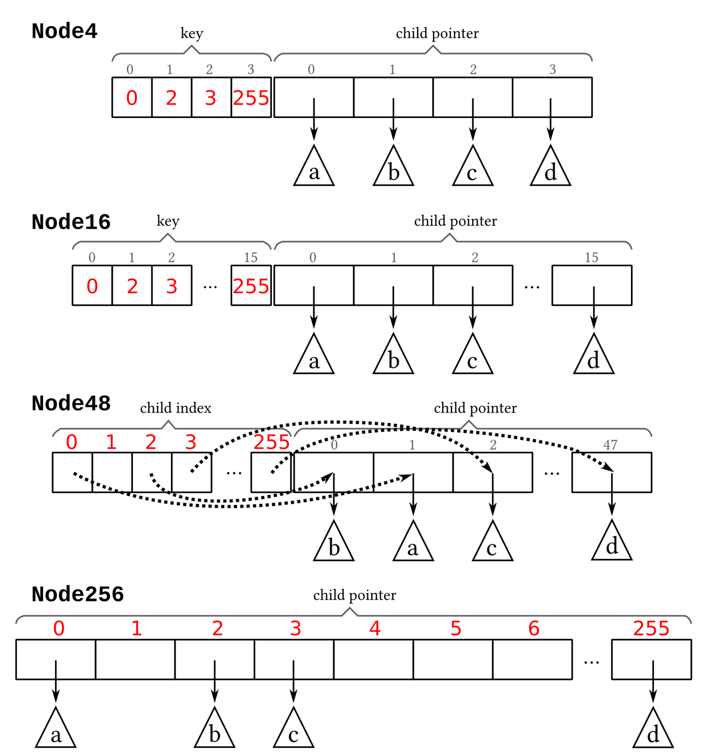
# Building a Concurrent Adaptive Radix Tree

Yuchen Liang, Hang Shu

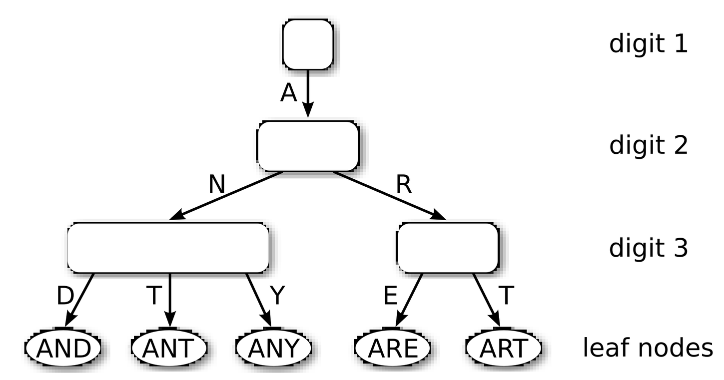
**Keywords** — Parallel Data Structures, Synchronizations

## I. Introduction

The Adaptive Radix Tree (ART) is an efficient in-memory index data structure on modern architectures that beats read-only search trees on lookup performance while at the same time supporting insertion and deletion [1,2] We implemented three synchronization protocols for the Adaptive Radix Tree and compared their performance under low and high contention. **Our results showed that the optimistic lock coupling approach scales better than the fine-grained lock coupling and the coarse-grained lock approach under both writer-only low contention scenarios and mixed writer and reader high contention scenarios.**




(a) Four different node types




(b) Adaptive-sized node in ART

Figure 1: ART with adaptive node representations (adopted from [1])

## II. Approach: Optimisitic Lock Coupling

Instead of preventing concurrent modification, we **optimistically assume that there will be no concurrent modification** and later use **version counters** to check if we need to restart the operation.

```
-------------------------------------------------------------
| version (bit 63-2) | lock (bit 1) | obsolete (bit 0) |
-------------------------------------------------------------
```

Listing 1: Internal layout of the optimistic lock.

When a writer is done with their operations, the unlock operation will clear the lock flag and increment the version counter. If a node is to be discarded from the tree by the writer, the obsolete flag will be marked. Optimistic lock works differently on readers. **Readers, however, do not acquire or release the lock.** Instead, before reading a node, the reader waits for the lock to be free and gets the current version of the counter. This version is kept during the operation and will later be checked against the latest version from the lock. If the two versions are not matched, the operation will restart.

```
bool LookupOLC(parent,node,key,
               depth,parent_version) {
  // <OLC>
  version = TryLockShared(node);
  TryUnlockShared(parent, parent_version);

  // common prefixes length
  p = node->PrefixMatches(key, depth);
  if (p != node->prefix.size()) {
    // prefix does not match
    // <OLC>
    TryUnlockShared(node, node_version);
    return false;
  }
  depth += p;
  if (IsLeaf(node)) {
    // <OLC>
    TryUnlockShared(node, node_version);
    return true;
  }
  CheckOrRestart(node, node_version);
  next = GetChild(node, key[depth]);
  if (next == nullptr) {
    // not found
    // <OLC>
    TryUnlockShared(node, node_version);
    return false;
  }
  return LookupOLC(node,next,key,
    depth+1,node_version);
}
```

(a) Lookup

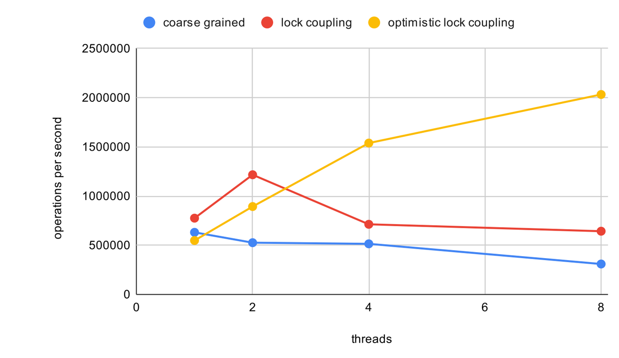
```
void InsertOLC(parent,node,key,leaf,depth,
               p_version,old_key) {
  node_version = TryLockShared(node);
  if (p != node->prefix.size()) {
    // prefix does not match
    TryUpgradeExclusive(parent,p_version);
    TryUpgradeExclusive(node,node_version,
                        parent);
    inner = MakeInner(&node->prefix, p);
    leaf_key = key[depth+p];
    inner_key = node->prefix[p];
    UnlockExclusive(inner);
    InsertChild(node, leaf_key, leaf);
    InsertChild(node, inner_key, inner);
    InsertChild(parent, old_key, node);
    UnlockExclusive(parent);
    return;
  }
  if (IsLeaf(node)) {
    TryUpgradeExclusive(node,node_version);
    TryUnlockShared(parent,p_version,node);
    Replace(node, leaf);
    UnlockExclusive(node);
    return;
  }
  depth += p;
  next = GetChild(node, key[depth]);
  // <OLC>
  CheckOrRestart(node, node_version);
  if (next == nullptr) {
    // If a grow needs to happen, we
    // will replace `node`, mark the
    // old one as obsolete.
    GrowInsert(node, leaf, depth);
    UnlockExclusive(parent);
    return;
  }
  TryUnlockShared(parent,p_version);
  InsertOLC(node, next, key, leaf, depth+1);
}
```

(b) Insert

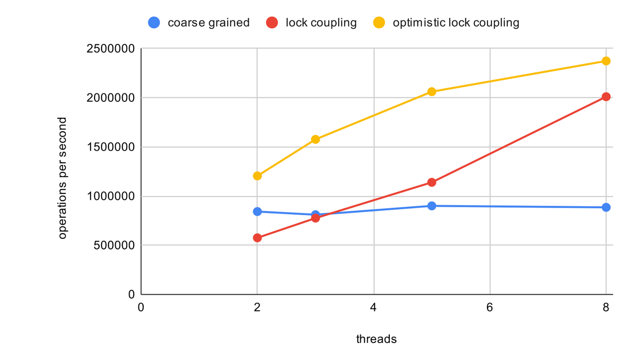
Figure 2: Pseudo code for lookup/insert operation synchronized using optimistic lock coupling.

## III. Results

To evaluate the performance of our concurrent ART implementation, we use **operations per second** as a measure of throughput, where an operation is counted by the completion of a single insert or contains call. We used integer keys for evaluation and ran our experiment on the GHC machines with varying thread counts. For the writes-only and reads-only experiments, the keys were generated with a **uniform distribution** such that accesses into the tree would have low contention. For the mixed experiment, the keys were generated with a **Zipfian** (skewed) distribution such that accesses into the tree would have high contention.




(a) Time constrained (10 seconds) scalability for write-only workload




(b) Time constrained (10 seconds) scalability for mixed workload

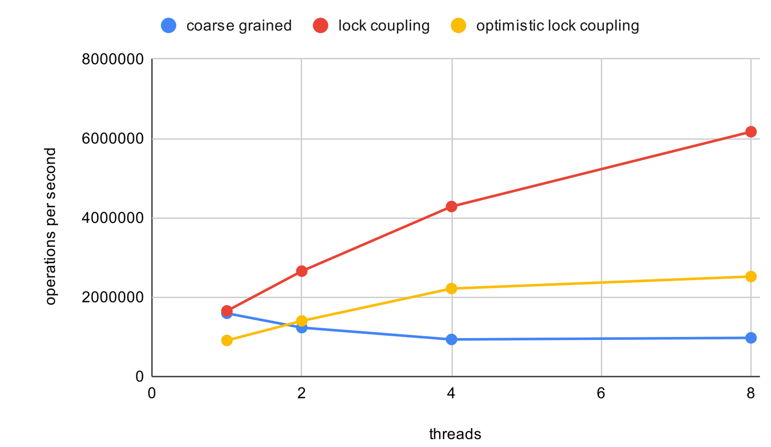



Figure 4: Time constrained (10 seconds) scalability for read-only workload

## IV. Key Takeaways

| % time | Function |
|---|---|
| 22.97 | `_Sp_counted_base<(gnu_cxx::_Lock_policy)2>::_M_release()` |
| 22.97 | `_Sp_counted_base<(gnu_cxx::_Lock_policy)2>::_M_add_ref_copy()` |
| 5.66 | `PrefixMatches(...)` |
| 2.23 | `OptimisticRWLock::AwaitUnlocked(unsigned long&)` |

Table 1: Time spent in various functions for **optimistic lock coupling** with 8 readers in the problem-constrained experiment (gprof results)

| % time | Function |
|---|---|
| 5.43 | `PrefixMatches(...)` |
| 2.36 | `std::__shared_mutex_pthread::lock_shared()` |
| 2.36 | `std::__shared_mutex_pthread::unlock_shared()` |

Table 2: Time spent in various functions for **lock coupling** with 8 readers in the problem-constrained experiment (gprof results)

The use of **shared pointers** is a bottleneck in our implementation of optimistic lock coupling as most of the time spent by OLC is done in accessing/updating a shared pointer's reference count.

## References

[1] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013. 38–49.

[2] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, 2016. 1–8.


{yuchenl3,hangshu}@andrew.cmu.edu