Building a Concurrent Adaptive Radix Tree

Yuchen Liang
yuchenl3@andrew.cmu.edu
Carnegie Mellon University

Hang Shu
hangshu@andrew.cmu.edu
Carnegie Mellon University

Summary

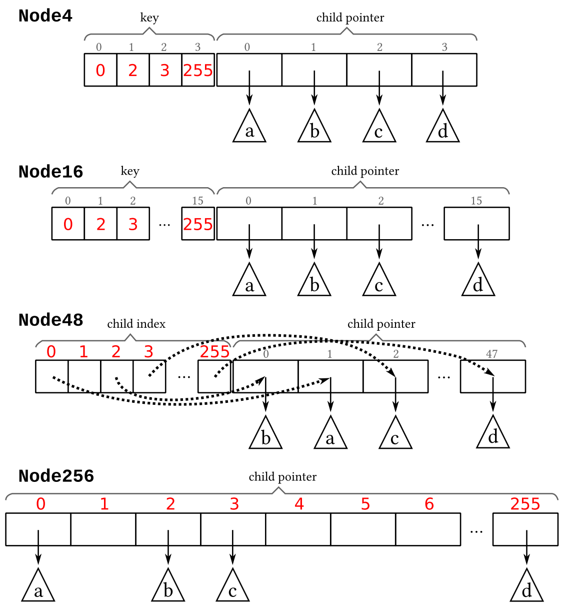
The Adaptive Radix Tree (ART) is an efficient in-memory index data structure on modern architectures that beats read-only search trees on lookup performance while at the same time supporting insertion and deletion [4]. We implemented three synchronization protocols for the Adaptive Radix Tree and compared 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.

Background

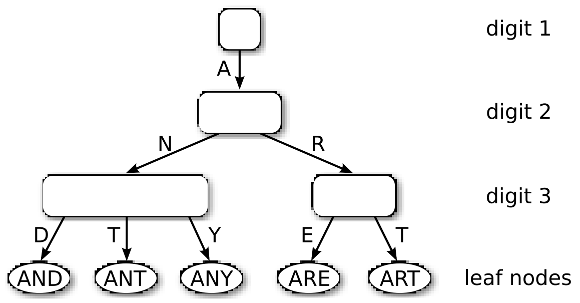
Relational database management systems provide indexes as a powerful tool to help users efficiently query data. In online transaction processing (OLTP) workloads, the execution plan generated from SQL queries usually involves a sequence of index operations. For main-memory database systems, it is essential to have good index performance as disk access is no longer the bottleneck. In a parallel architecture, it is critical to have an efficient synchronization approach so that the performance of the data structure scales with the number of processors. Holding a lock over the entire data structure is not the solution as it would serialize all operations and greatly limit the system's throughput. The Adaptive Radix Tree was originally designed as an efficient single-threaded index structure for main-memory database management systems and then parallelized to be efficient in parallel systems [4,5]. The following section describes the ART data structure and breaks down the typical workload.

The Adaptive Radix Tree

The Adaptive Radix Tree belongs to the radix tree family. Unlike n-ary search trees and hash tables, radix trees use the digital representation of the keys to traverse the tree instead of comparing the hashes or the keys themselves. As a result, all operations have $O(k)$ complexity where k is the length of the keys. For the specific use as an index structure, the number of keys would be much greater than the key length such that radix trees become very efficient. The ART data structure, however, distinguishes itself from other radix trees by choosing its internal node representation dynamically. The four different node representations are shown in Figure 1.



(a) Four different node types



(b) Adaptive-sized node in ART

Figure 1: ART with adaptive node representations (adopted from [4])

When a node is first created, it will use `Node4`, the smallest representation, and later grow and shrink to other representations when necessary. The ART data structure also introduces *path compression* optimization, meaning that a child node will be collapsed into its parent if the parent node has only a single child. To implement this, each node additionally stores a prefix of key bytes in the node header, which is illustrated in . With this optimization, we significantly reduce the height of the tree. With adaptive node sizes and path compression, the ART data structure reduces its space consumption while at the same time maintaining high performance. However, these additional optimizations do make ART harder to synchronize compared to other radix tree variants, which we will discuss further in the **Approach** section.

Typical workload

As shown Listing 1, the ART data structure can support lookup, insert, remove, range scan, and prefix match operations. In our implementation, we focused on synchronizing lookup and insert operations. A normal workload on the ART data structure would involve multiple workers performing read/write operations on the data structure concurrently. Our synchronization protocols on ART ensure reader-writer exclusion and writer-writer exclusion to guarantee correctness and memory safety.

```
template <typename K, typename V, typename P = Prefix<K>>
class AdaptiveRadixTree {

  // APIs with synchronization implemented.
  bool Contains(K& key);
  void Insert(K& key, V& value);

  // Other APIs ART could support.
  // void Remove(K& key);
  // void Scan(K& key_begin, K& key_end) -> std::vector<V>;
  // void BeginWith(P& prefix) -> std::vector<V>;
};
```

Listing 1: APIs supported by Adaptive Radix Tree.

Approach

Our implementations of the concurrent Adaptive Radix Tree are written in C++ and target the CPU architecture. The workers doing read/write operations on the data structure are each mapped to a CPU thread, and the workers synchronize with each other under the shared-memory model. We first implemented a sequential version of the Adaptive Radix Tree by following the pseudo-code described in the paper [4]. We then implemented three synchronization protocols for the Adaptive Radix Tree (1) coarse-grained locking, (2) fine-grained lock coupling, and (3) optimistic lock coupling. We will omit the discussion of the coarse-grained locking, as this approach takes a giant lock that essentially makes accesses into ART serial. The following sections will focus on lock coupling and optimistic lock coupling.

Lock coupling

We first tried to implement lock coupling on the Adaptive Radix Tree. The basic idea is to hold at most two locks at a time going from the root of the tree. This is possible as a modification in ART will at most affect 2 nodes: the node where the modification occurs, and the parent of the node when a grow operation is needed during insertion. Compared to ART, the changes in a n-ary tree may potentially propagate to the root as a result of re-balancing. Under this approach, we add a lock to each node in the tree and use the lock to ensure access to the node is thread-safe. To achieve better parallelism, we chose to use the `std::shared_mutex` a reader-writer lock based on the `pthread` library so that multiple readers can read a node at the same time. The pseudo-code is shown in Figure 9 of the appendix, and the changes required by lock coupling are labeled with `<LC>`. In terms of implementation effort, this approach requires the least amount of code changes from our serial code. However, as we show in the **Results** section, acquiring and releasing the lock could cause cache misses that invalidate cache lines on all threads.

Optimistic lock coupling

The next synchronization approach we chose is optimistic 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. We will first describe our implementation of the optimistic lock, and then discuss how it is used on the Adaptive Radix Tree.

Optimistic lock implementation

Internally the optimistic lock consists of a lock and a version counter. All the content is packed into a 64-bit integer stored as `std::atomic<uint64_t>`. As shown in Listing 2, we use bit `0` as the obsolete flag and bit `1` as the lock flag. The top 62 bits are used as the version counter. For writers, the optimistic lock works very similar to normal locks which provides mutual exclusion by writing to the lock flag. 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. With upgrade operation also supported by our lock, the writers can first take a shared lock and upgrade it to exclusive when it is necessary to modify the node. This also helps improve performance in a concurrent setting.

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

Listing 2: Internal layout of the optimistic lock.

The optimistic lock works really well in modern parallel CPU architecture. Since readers do not write to shared memory locations when they acquire the lock, we eliminated a lot of cache traffic that would otherwise needed for traditional lock coupling. We acknowledge that our optimistic lock implementation does not guarantee fairness, but it does provide mutual exclusion and makes sure at least some thread is making progress.

Optimistic lock coupling on ART

When using optimistic lock coupling to synchronize the Adaptive Radix Tree, we have an optimistic lock API similar to the normal lock API, except it is possible for certain lock and unlock operations to fail. We also added version checks before accessing the next child node to prevent dereferencing invalid pointers. As mentioned in the previous section, when a writer removes a node from the tree, it will not immediately free the node but instead mark the node as obsolete. Thus, some memory reclamation procedure is needed for us to free to node. We decided to use `std::shared_ptr` to manage the lifetime of the node. It turns out to not be a good choice, as we will discuss in the results section. We did consider bringing in other libraries [1] that are specially designed to do efficient deferred memory reclamation in concurrent settings, but limitations in the library make it incompatible with our use case. Specifically, the library currently does not support aliasing on their smart pointer types, but this is an important feature for us as we have multiple node types. We also show the pseudo-code for implementing optimistic lock coupling for insert and lookup operations in Figure 10 of the appendix.

Results

To evaluate the performance of our concurrent ART implementation, we use operations per second as a measure of throughput. An operation is counted by the completion of a single insert or contains call. We ran our experiment on the GHC machines with varying thread counts.

We choose to evaluate the performance on integer keys (`uint64_t`). All the keys queried or inserted into the Adaptive Radix Tree will be in the key space from 0 to 10000000. The keys are generated through either a uniform distribution, or a Zipfian (skewed) distribution to simulate low and high contention scenarios. We run two sets of experiments, one set measures time-constrained scaling and the other measures problem-constrained scaling.

Time constrained experimental setup

In the time-constrained experiments, the duration of our program is fixed at 10 seconds. We have a variable amount of reader threads calling the contains function and a variable amount of writer threads calling the insert function. In each of the experiments, we first insert 1/16 of our total key space (625000 keys) into the Adaptive Radix Tree. In the readers-only experiment (Figure 2), we generate the keys using a uniform distribution and increase the number of reader threads while keeping the number of writer threads to 0. In the writers-only experiment (Figure 3), we generate the keys using a uniform distribution and increase the number of writer threads while keeping the number of reader threads to 0. In the mixed experiment (Figure 4), we generate the keys using a Zipfian distribution and fix the number of writer threads to 1, while increasing the number of reader threads.

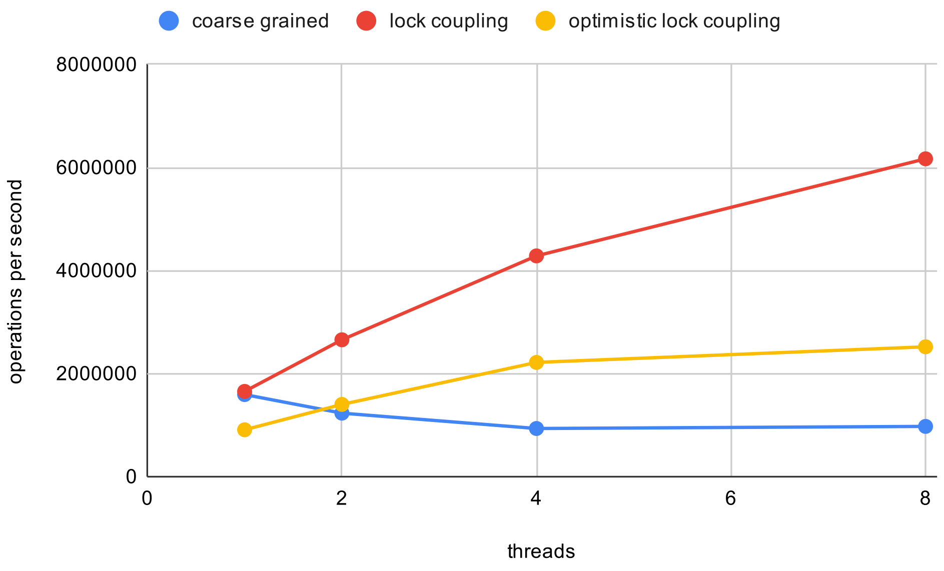


Figure 2: Time constrained (10 seconds) scalability for read-only workload

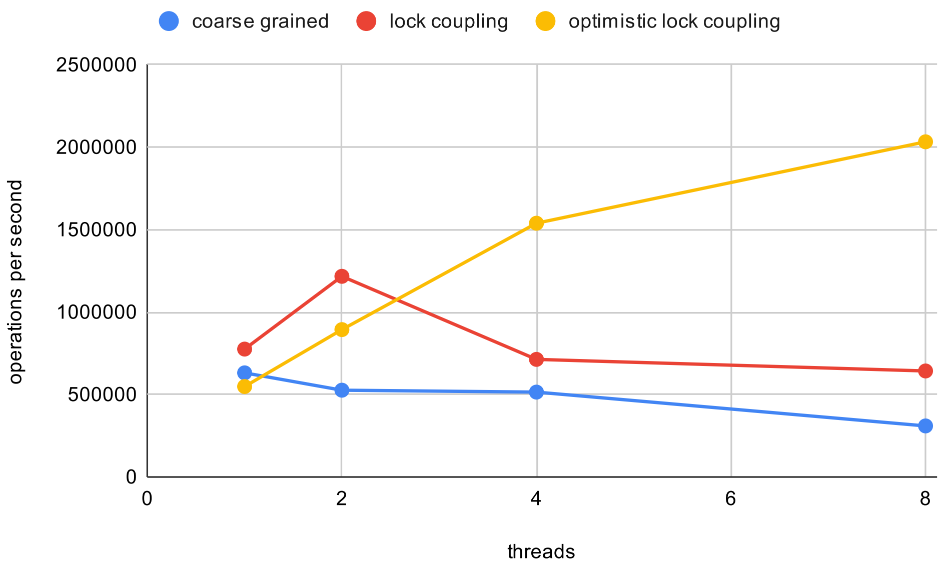


Figure 3: Time constrained (10 seconds) scalability for write-only workload

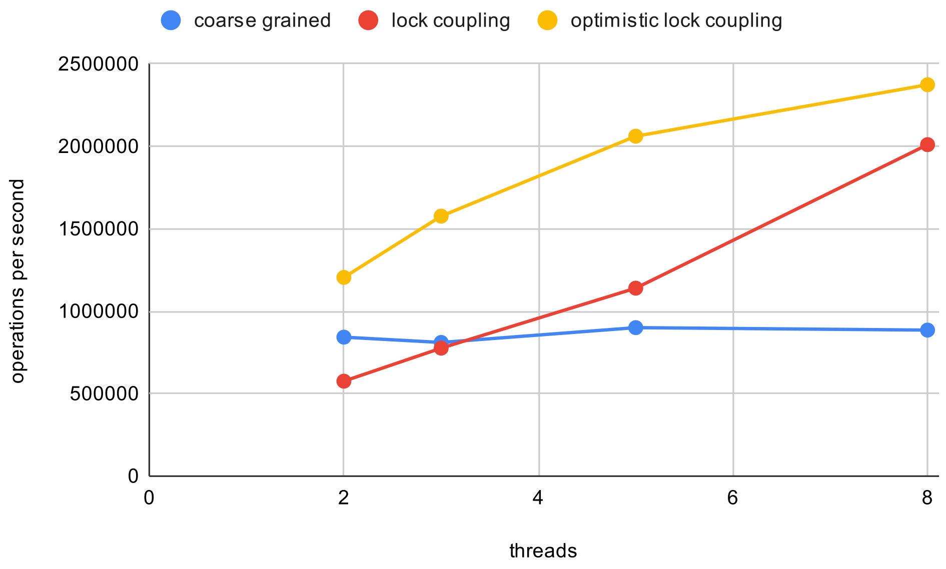


Figure 4: Time constrained (10 seconds) scalability for mixed workload

Problem constrained experimental setup

In the problem-constrained experiments, the number of operations is fixed to be 10 million operations, and we have a variable amount of reader threads calling the contains function and a variable amount of writer threads calling the insert function. In each of the experiments, we first insert 1/16 of our total key space (625000 keys) into the Adaptive Radix Tree. In the readers-only experiment (Figure 2), we generate the keys using a uniform distribution and increase the number of reader threads while keeping the number of writer threads to 0. In the writers-only experiment (Figure 3), we generate the keys using a uniform distribution and increase the number of writer threads while keeping the number of reader threads to 0. In the mixed experiment (Figure 4), we generate the keys using a Zipfian distribution and fix the number of writer threads to 1, while increasing the number of reader threads. In the mixed experiment, the writer thread will insert a total number of operations / total number of threads amount of keys, while the readers will evenly split the remaining number of keys to read.

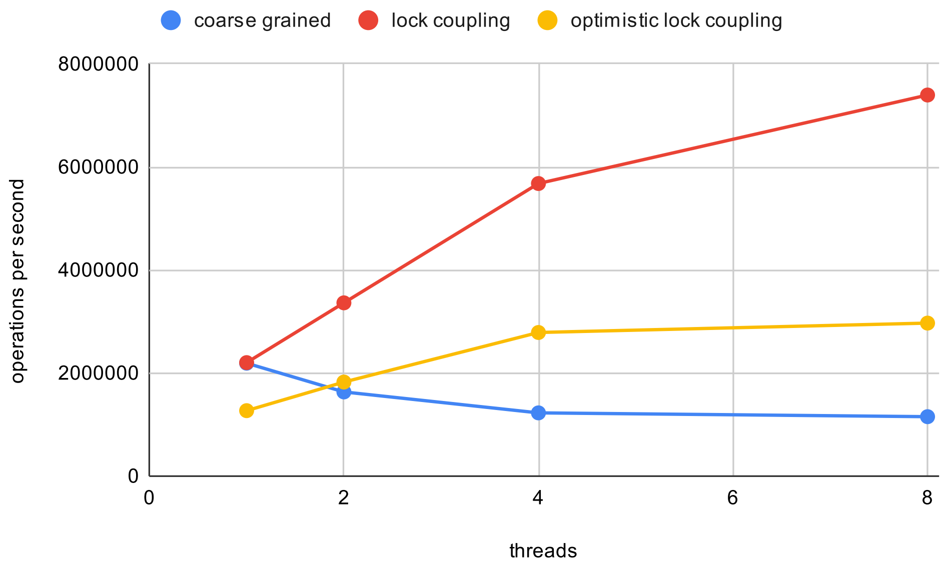


Figure 5: Problem constrained (10 million operations) scalability for read-only workload

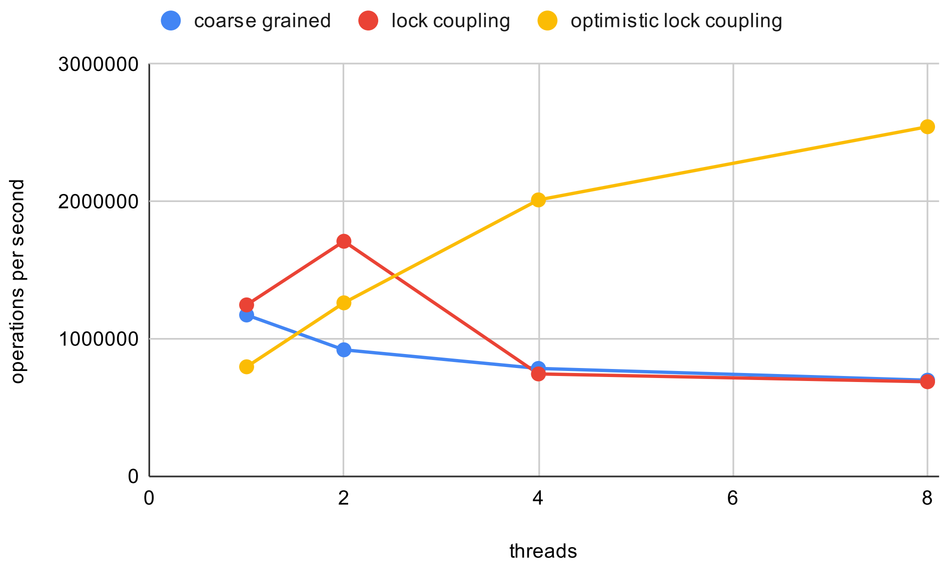


Figure 6: Time constrained (10 million operations) scalability for write-only workload

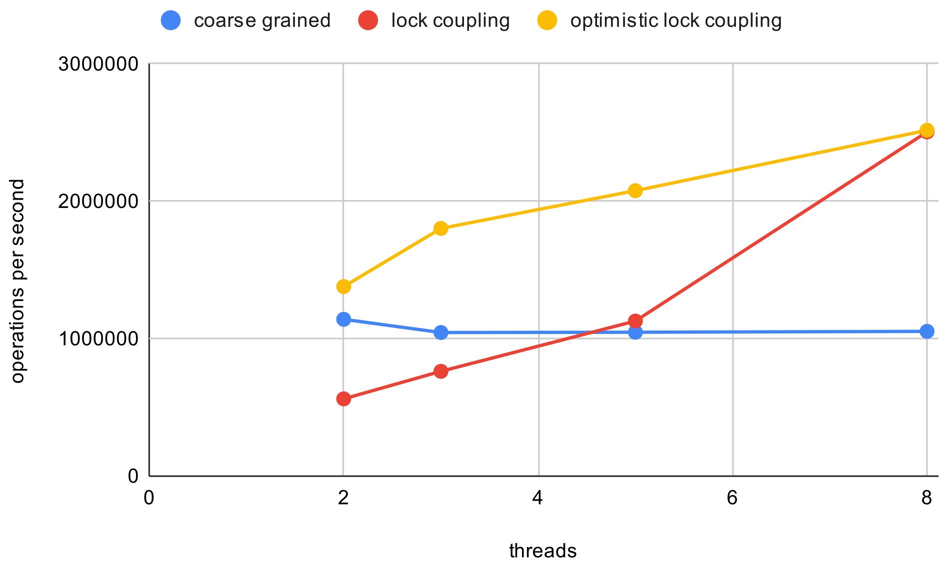


Figure 7: Time constrained (10 million operations) scalability for mixed workload

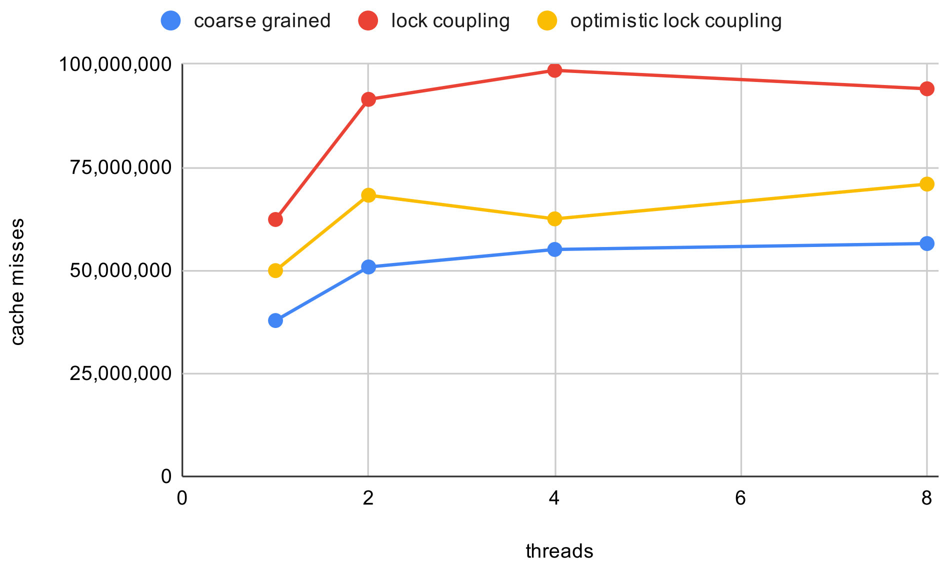


Figure 8: Problem constrained (10 million operations) cache misses for read-only workload

Analysis

The speedup we see from the graphs is not linear: at best, we see a speedup of around 3.71x from optimistic lock coupling from 1 to 8 threads in the writes-only experiments (Figure 3 and Figure 6) and a speedup of around 3.73x from lock coupling from 1 to 8 threads in the reads only experiments (Figure 2 and Figure 5). A major contributing factor to this is due to the synchronization overhead from reader and writer threads and/or writer and writer threads. Since there is a high level of contention between the threads attempting to acquire the locks protecting these nodes, we see the number of cache misses increasing with the number of threads (Figure 8) as when a thread does grab a hold of a lock, it invalidates all other cache lines holding this lock.

From the following figures, we see that optimistic lock coupling generally performs the best, however, for read-only applications, lock coupling performs better than optimistic lock coupling. We expect optimistic lock coupling to perform better than lock coupling for a few reasons. First, the optimistic reader lock does not invalidate the other cache lines holding the lock as it simply does a read instruction to check the value of the lock bit, while in the lock coupling approach, the reader-writer locks do invalidate the other cache lines holding the lock. Second, in optimistic lock coupling, a write operation (insert) does not hold a write lock until it is necessary, i.e., for an insert function, the traversing down the nodes will be guarded with optimistic read locks until it reaches a node that it needs to write to in which it will attempt to grab hold of a write lock. For lock coupling, an insert call will always grab a write lock, causing other reader/writer threads traversing down the same path to wait on this writer thread to finish with the current nodes that are on. We can see from Figure 4, Figure 3, Figure 7, and Figure 6 that where there is at least a single thread writing to the tree, optimistic lock coupling does perform the best.

However, from Figure 2 and Figure 5 (the read-only experiments), we see lock coupling performing better than optimistic lock coupling. This comes from a design issue within our implementation of optimistic lock coupling stemming from our use of shared pointers. In optimistic lock coupling for our contains function, we are holding a copy of a shared pointer to a node instead of a reference, while in lock coupling, we are holding a reference of a shared pointer to a node. This is because, with optimistic lock coupling, a writer can delete/update a node, but the reader's pointer to the deleted/updated node is still valid. In lock coupling, this situation is not possible, as the writer would have to wait for the reader to complete its operation and unlock.

Our use of shared pointers in optimistic lock coupling causes a case where a thread can be accessing a shared pointer at the same time another thread is updating that shared pointer (writer grows/shrinks a node, meaning that the pointer to this node is replaced by another pointer, at the same time, the reader is reading the node the writer is updating). To eliminate this race condition, we needed to make loads and stores of shared pointers in optimistic lock coupling atomic. The current atomic loads and stores implementation for `shared_ptr` in the standard library, however, uses a mutex to wrap around the loads and stores of shared pointers. This can cause contention and potentially lead to more cache misses due to cache invalidation necessary to lock or unlock a mutex. Another issue with using copies of a shared pointer is that shared pointers keep track of a reference count of how many threads have copies of this shared pointer. This means that when a thread creates a copy of a shared pointer or the shared pointer goes out of scope of the thread, the reference count is incremented or decremented, and since this shared pointer is shared amongst different threads thereby different cores, this can also lead to more cache invalidation.

With the gprof profiler, we see that optimistic lock coupling spends most of its time attempting to access the reference count from the first two rows in Table 1.

Percent time spent	**Function**
22.97	`std::_Sp_counted_base<(__gnu_cxx::_Lock_policy)2>::_M_release()`
22.97	`std::_Sp_counted_base<(__gnu_cxx::_Lock_policy)2>::_M_add_ref_copy()`
5.66	`cart::olc::Node<unsigned long, unsigned long>::PrefixMatches(...)`
2.23	`cart::olc::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)

Percent time spent	**Function**
7.79	`std::__shared_ptr_access<cart::lc::Node<un... (...:_M_get() const`
7.61	`std::__shared_ptr_access<cart::lc::Node<...>,...>::operator->() const`
6.52	`std::__shared_ptr<cart::lc::Node<...>,...>::get() const`
5.43	`cart::olc::Node<unsigned long, unsigned long>::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)

To improve upon the optimistic lock coupling design, we would have to replace our usage of shared pointers with raw pointers so that we can avoid cache invalidation due to updating the reference count. We would also not need the mutex to protect the loading and storing of pointers since loading and storing a raw pointer is atomic. However, this change from shared pointers to raw pointers would require us to manually garbage-collect pointers to nodes that are about to be removed from the tree. The other option is to contribute to the `concurrent_deferred_rc` library and support aliasing so the smart pointer types can be used by our implementation [1]. This library provides smart-pointer semantics but can make use of efficient memory reclamation algorithms (e.g. epoch-based reclamation [3], hazard pointers [6], and Hyaline [7]) to perform garbage collections. To further improve the performance of ART on modern architecture, it is likely we will need to use arena-based memory allocators like `jemalloc` makes memory allocation scalable on multiprocessor machine [2].

The choice of machine target was sound, as we would not have been able to use a GPU for our application was our program had lots of data dependencies (parent to children node structure), meaning that it would not fit the data-parallel model that GPUs are useful for.

References

[1] Daniel Anderson, Guy E Blelloch, and Yuanhao Wei. 2021. Concurrent deferred reference counting with constant-time overhead. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021. 526–541.

[2] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[3] Keir Fraser. 2004. *Practical lock-freedom.*

[4] 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.

[5] 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.

[6] Maged M Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 491–504.

[7] Ruslan Nikolaev and Binoy Ravindran. 2019. Hyaline: fast and transparent lock-free memory reclamation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019. 419–421.

Work Distribution

Work done by Yuchen Liang

- Serial implementation of the Adaptive Radix Tree.
- Optimistic Lock Implementation
- Optimistic Lock Coupling implementation

Work done by Hang Shu

- Benchmark implementation
- Lock Coupling implementation

Appendix A: Pseudo-code

```
bool LookupLC(parent,node,key,depth)
{
  RWLockShared(node); // <LC>
  RWUnlockShared(parent); // <LC>

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

(a) Lookup

```
void InsertLC(parent,node,key,leaf,depth)
{
  RWLockExclusive(node); // <LC>
  if (p != node->prefix.size()) {
    // prefix does not match
    inner = MakeInner(&node->prefix, p);
    leaf_key = key[depth+p];
    inner_key = node->prefix[p];
    swap(node, inner);
    RWUnlockExclusive(inner); // <LC>
    InsertChild(node, leaf_key, leaf);
    InsertChild(node, inner_key, inner);
    RWUnlockExclusive(parent); // <LC>
    return;
  }
  if (IsLeaf(node)) {
    Replace(node, leaf);
    RWUnlockExclusive(node); // <LC>
    RWUnlockExclusive(parent); // <LC>
    return;
  }
  depth += p;
  next = GetChild(node, key[depth]);
  if (next == nullptr) {
    GrowInsert(node, leaf, depth);
    RWUnlockExclusive(node); // <LC>
    RWUnlockExclusive(parent); // <LC>
    return;
  }
  RWUnlockExclusive(parent); // <LC>
  InsertLC(node, next, key, leaf, depth+1);
}
```

(b) Insert

Figure 9: Pseudo code for lookup/insert operation synchronized using lock coupling.

```
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 10: Pseudo code for lookup/insert operation synchronized using optimistic lock coupling.