The ART of Practical Synchronization

Viktor Leis Florian Scheibner Alfons Kemper Thomas Neumann
Technische Universität München
{leis,scheibnf,kemper,neumann}@in.tum.de

ABSTRACT

The performance of transactional database systems is critically dependent on the efficient synchronization of in-memory data structures. The traditional approach, fine-grained locking, does not scale on modern hardware. Lock-free data structures, in contrast, scale very well but are extremely difficult to implement and often require additional indirections. In this work, we argue for a middle ground, i.e., synchronization protocols that use locking, but only sparingly. We synchronize the Adaptive Radix Tree (ART) using two such protocols, Optimistic Lock Coupling and Read-Optimized Write EXclusion (ROWEX). Both perform and scale very well while being much easier to implement than lock-free techniques.

1. INTRODUCTION

In traditional database systems, most data structures are protected by fine-grained locks[1]. This approach worked well in the past, since these locks were only acquired for a short time and disk I/O dominated overall execution time. On modern servers with many cores and where most data resides in RAM, synchronization itself often becomes a major scalability bottleneck. And with the increasing number of CPU cores—Intel's current server platform Broadwell EP supports up to 24 cores per socket—efficient synchronization will become even more important.

Figure 1 gives an overview over the synchronization paradigms discussed in this paper. Besides traditional fine-grained locking, which is known to scale badly on modern CPUs, the figure shows the lock-free paradigm, which offers strong theoretical guarantees, and Hardware Transactional Memory (HTM), which requires special hardware support. When designing a data structure, so far, one had to decide between the extreme difficulty of the lock-free approach, special hardware support of HTM, and poor scalability of locking. In this paper, we present two additional points in the design space that fill the void in between. *Optimistic Lock Coupling* and *ROWEX* are much easier to use than lock-free synchronization but offer similar scalability without special hardware support.

[1] In this paper, we always use the term "lock" instead of "latch" since we focus on low-level data structure synchronization, not high-level concurrency control.

DaMoN'16, June 26-July 01 2016, San Francisco, CA, USA

 ISBN 978-1-4503-4319-0/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2933349.2933352

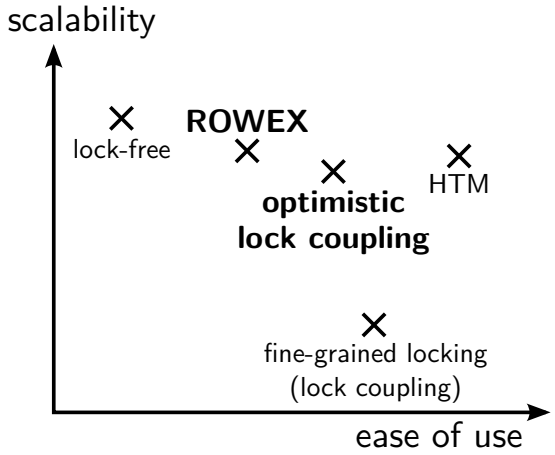

Figure 1: Overview of synchronization paradigms

We focus on synchronizing the *Adaptive Radix Tree (ART)* [12], a general-purpose, order-preserving index structure for main-memory database systems. ART is an interesting case study, because it is a non-trivial data structure that was *not* designed with concurrency in mind rather with high single-threaded performance. We present a number of synchronization protocols for ART and compare them experimentally.

Our main goal in this paper, however, is to distill general principles for synchronizing data structures in general. This is important for two reasons. First, besides index structures, database systems also require other data structures that must be concurrently accessible like tuple storage, buffer management data structures, job queues, etc. Second, concurrent programs are very hard to write and even harder to debug. We therefore present our ideas, which, as we discuss in Section 6, are not entirely new, as general building blocks that can be applied to other other data structures.

The rest of this work is organized as follows. We first present necessary background about the Adaptive Radix Tree in Section 2. The two new synchronization paradigms *Optimistic Lock Coupling* and *Read-Optimized Write EXclusion* are introduced in Section 3 and Section 4. Section 5 evaluates the presented mechanisms. Finally, after discussing related work in Section 6, we present our conclusions in Section 7.

2. THE ADAPTIVE RADIX TREE (ART)

Trie data structures, which are also known as radix trees and prefix trees, have been shown to outperform classical in-memory search trees [12, 10]. At the same time, and in contrast to hash

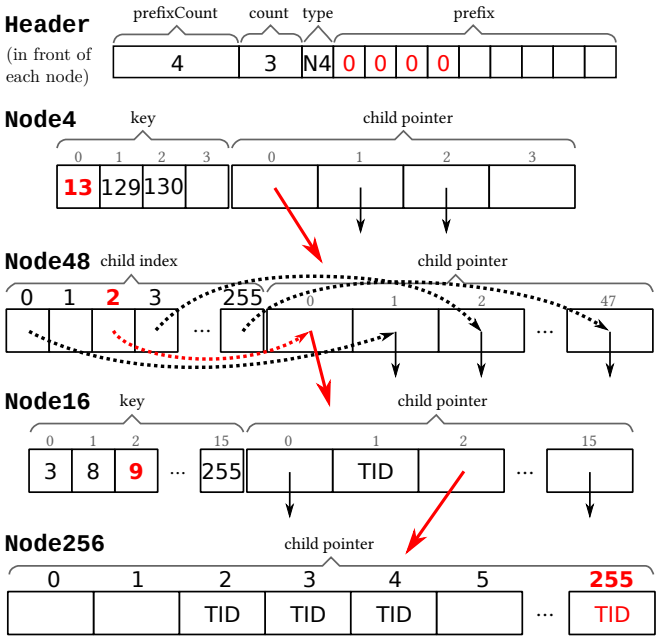

Figure 2: The internal data structures of ART

tables, they are order-preserving, making them very attractive indexing structures for main-memory database systems.

What distinguishes ART [12] from most other trie variants is that it uses an **adaptive node structure**. ART dynamically chooses the internal representation of each node from multiple data structures. The four available node types are illustrated in Figure 2. Initially, the smallest node type (`Node4`) is selected, and, as entries are inserted into that node, it is replaced with a larger node type. In the figure, if two more entries would be inserted into the (`Node4`), which currently holds 3 entries, it would be replaced by a (`Node16`).

Another important feature of ART is **path compression**, which collapses nodes with only a single child pointer into the first node with more than one child. To implement this, each node stores a *prefix* of key bytes in its header. This allows indexing long keys (e.g., strings) effectively, because the optimization significantly reduces the height of the tree. The example header shown in Figure 2 stores a prefix of 4 zero bytes thus reducing the height by 4 levels.

In HyPer, where ART is the default indexing structure [8], ART maps arbitrary keys to **tuple identifiers (TIDs)**. As the figure shows, the TIDs are stored directly inside the pointers. Pointers and TIDs are distinguished using "pointer tagging", i.e., by using one of the pointer's bits.

Adaptive node types and path compression reduce space consumption while allowing for nodes with high fanout and thus ensure high overall performance. At the same time, these features are also the main challenges for synchronizing ART. Tries without these features are easier to synchronize, which makes ART a more interesting case study for synchronizing non-trivial data structures.

3. OPTIMISTIC LOCK COUPLING

Lock coupling [1], i.e., holding at most 2 locks at a time during traversal, is the standard method for synchronizing B-trees. One interesting property of ART is that modifications affect at most two nodes: the node where the value is inserted or deleted, and potentially its parent node if the node must grow (during insert) or shrink (during deletion). In contrast to B-trees, a modification will never propagate up to more than 1 level. Lock coupling can therefore be applied to ART even more easily than to B-trees.

The left-hand-side of Figure 3 shows the necessary changes for synchronizing the lookup operation of ART with lock coupling. The pseudo code uses read-write locks to allow for concurrent readers. Insert and delete operations can also initially acquire read locks before upgrading them to write locks if necessary. This allows to avoid exclusively locking nodes near the root, which greatly enhances concurrency because most updates only affect nodes far from the root.

Lock coupling is simple and seems to allow for a high degree of parallelism. However, as we show in Section 5, it performs very badly on modern multi-core CPUs even if the locks do not logically conflict at all, for example in read-only workloads. The reason is that concurrent locking of tree structures causes many unnecessary cache misses: Each time a core acquires a read lock for a node (by writing to that node), all copies of that cache line are invalidated in the caches of all other cores. Threads, in effect, "fight" for exclusive ownership of the cache line holding the lock. The root node and other nodes close to it become contention points. Therefore, other synchronization mechanisms are necessary to fully utilize modern multi-core CPUs.

Optimistic Lock Coupling is similar to "normal" lock coupling, but offers dramatically better scalability. Instead of preventing concurrent modifications of nodes (as locks do), the basic idea is to optimistically assume that there will be no concurrent modification. Modifications are detected after the fact using version counters, and the operation is restarted if necessary. From a performance standpoint, this optimism makes a huge difference, because it dramatically reduces the number of writes to shared memory locations.

3.1 Optimistic Locks

Figure 3 compares the pseudo code for lookup in ART using lock coupling and Optimistic Lock Coupling (the code for insert can be found in Appendix B). Clearly, the two versions are very similar. The difference is encapsulated in the `readLockOrRestart` and `readUnlockOrRestart` functions, which mimic a traditional locking interface but are implemented differently. This interface makes it possible for the programmer to reason (almost) as if she was using normal read-write locks.

Internally, an *optimistic lock* consists of a lock and a version counter. For writers, optimistic locks work mostly like normal locks, i.e., they provide mutual exclusion by physically acquiring (by writing into) the lock. Additionally, each `writeUnlock` operation causes the version counter associated with the lock to be incremented. Read operations, in contrast, do not actually acquire or release locks. `readLockOrRestart` merely waits until the lock is free, before returning the current version. `readUnlockOrRestart`, which takes a version as an argument, makes sure the lock is still free and that the version (returned by `readLockOrRestart`) did not change. If a change occurred, the lookup operation is restarted from the root of tree. In our implementation we encode the lock and the version counter (and an obsolete flag) in a single 64 bit word that is updated atomically and is stored in the header of each ART node. A full description of the optimistic lock primitives can be found in Appendix A.

Let us mention that, for debugging purposes, it is possible to map the optimistic primitives to their pessimistic variants (e.g., `pthread_rwlock`). This enables thread analysis tools like `Helgrind` to detect synchronization bugs. Version/lock combinations have been used in the past for synchronizing data structures (e.g., [4, 2, 18]), usually in combination with additional, data structure-specific tricks that reduce the number of restarts. Opti-

```
lookup(key, node, level, parent)
   readLock(node)
   if parent != null
      unlock(parent)
   // check if prefix matches, may increment level
   if !prefixMatches(node, key, level)
      unlock(node)
      return null  // key not found
   // find child
   nextNode = node.findChild(key[level])

   if isLeaf(nextNode)
      value = getLeafValue(nextNode)
      unlock(node)
      return value  // key found
   if nextNode == null
      unlock(node)
      return null  // key not found
   // recurse to next level
   return lookup(key, nextNode, level+1, node)
```

```
1  lookupOpt(key, node, level, parent, versionParent)
2     version = readLockOrRestart(node)
3     if parent != null
4        readUnlockOrRestart(parent, versionParent)
5     // check if prefix matches, may increment level
6     if !prefixMatches(node, key, level)
7        readUnlockOrRestart(node, version)
8        return null  // key not found
9     // find child
10    nextNode = node.findChild(key[level])
11    checkOrRestart(node, version)
12    if isLeaf(nextNode)
13       value = getLeafValue(nextNode)
14       readUnlockOrRestart(node, version)
15       return value  // key found
16    if nextNode == null
17       readUnlockOrRestart(node, version)
18       return null  // key not found
19    // recurse to next level
20    return lookupOpt(key, nextNode, level+1, node, version)
```

Figure 3: Pseudo code for a lookup operation that is synchronized using lock coupling (left) vs. Optimistic Lock Coupling (right). The necessary changes for synchronization are highlighted

mistic Lock Coupling is indeed a very general technique that allows consistent snapshots over multiple nodes (e.g., 2 at a time for lock coupling). The region can even involve more than 2 nodes, but should obviously be as small as possible to reduce conflicts.

3.2 Assumptions of Optimistic Lock Coupling

To make Optimistic Lock Coupling work correctly, there are certain properties that an algorithm must fulfill. After `readLockOrRestart` has been called, a reader may see intermediate, inconsistent states of the data structure. An incorrect state will be detected later by `readUnlockOrRestart`, but, in some cases, can still cause problems. In particular, care must be taken to avoid (1) infinite loops and (2) invalid pointers. For ART, an infinite loop cannot occur. To protect against invalid pointers, it is necessary to add an additional version check (line 11 in Figure 3). Without this check, `nextNode` might be an invalid pointer, which may cause the program to crash. These additional checks are needed before an optimistically read pointer is dereferenced.

Another aspect that needs some care is deletion of nodes. A node must not be immediately reclaimed after removing it from the tree because readers might still be active. We use epoch-based memory reclamation to defer freeing such nodes. Additionally, when a node has been logically deleted, we mark it as *obsolete* when unlocking the node (cf. `writeUnlockObsolete` in Appendix A). This allows other writers to detect this situation and trigger a restart.

One theoretical problem with Optimistic Lock Coupling is that—in pathological cases—a read operation may be restarted repeatedly. A simple solution for ensuring forward progress is to limit the number of optimistic restarts. After this limit has been reached, the lookup operation can acquire write locks instead.

To summarize, Optimistic Lock Coupling is remarkably simple, requires few changes to the underlying data structure, and, as we show in Section 5, performs very well. The technique is also quite general and can be applied to other data structures (e.g., B-trees).

4. READ-OPTIMIZED WRITE EXCLUSION

Like many optimistic concurrency control schemes, Optimistic Lock Coupling performs very well as long as there are few conflicts. The big disadvantage is, however, that all operations may restart. Restarts are particularly undesirable for reads, because they are predominant in many workloads. We therefore present a second synchronization technique that still uses locks for writes, but where reads always succeed and never block or restart. We call this technique *Read-Optimized Write EXclusion (ROWEX)*.

4.1 General Idea

ROWEX is a synchronization paradigm that lies between traditional locking and lock-free techniques (cf. Figure 1). It provides the guarantee that reads are non-blocking and always succeed. Synchronizing an existing data structure with ROWEX is harder than with (optimistic) lock coupling, but generally still easier (i.e., at least possible) than designing a similar lock-free data structure. The main tool of ROWEX is the write lock, which provides additional leverage in comparison with lock-free approaches that must confine themselves with primitive atomic operations like compare-and-swap. The write locks are acquired only infrequently by writers and never by readers.

The basic idea with ROWEX is that, before modifying a node, the lock for that node is first acquired. This lock only provides exclusion relative to other writers, but not readers, which never acquire any locks (and never check any versions). The consequence is that writers must ensure that reads are always consistent by using atomic operations. As we describe in the following, ROWEX generally requires some modifications to the internal data structures.

Although the name ROWEX indicates that reads are fast, writes are not slow either (despite requiring locks). The reason is that writers only acquire local locks at the nodes that are actually changed, i.e., where physical conflicts are likely. Even lock-free designs will often have cache line invalidations that impair scalability when writing to the same node. ROWEX thus provides very good scalability while still being realistically applicable to many existing data structures.

4.2 ROWEX for ART

One important invariant of ART is that every insertion/deletion order results in the same tree because there are no rebalancing operations. Each key, therefore, has a deterministic physical location. In B-link trees [11], in contrast, a key may have been moved to a node on the right (and never to the left due to deliberately omitting underflow handling).

To synchronize ART using ROWEX, we first discuss local modifications within the 4 node types before describing how node replacement and finally path compression are handled.

To allow concurrent **local modifications**, accesses to fields that may be read concurrently must be atomic. These fields include the key bytes (in `Node4` and `Node16`), the child indexes (in `Node48`), and the child pointers (in all node types). In C++ 11 this is done by using the `std::atomic` type, which ensures that appropriate memory barriers are inserted[2]. These changes are already sufficient for the `Node48` and `Node256` types and allow adding (removing) children to (from) these nodes. For the linear node types (`Node4` and `Node16`), which are structurally very similar to each other, some additional conceptual changes are necessary. In the original design, the keys in linear nodes are sorted (cf. Figure 2) to simplify range scans. To allow for concurrent modifications while reads are active, we avoid maintaining the sorted order and append new keys at the end of the node instead. Deletions simply set the child pointer to null, and slots are reclaimed lazily by replacing the node with a new one. With this change, lookups must check all keys (i.e., 4 for `Node4` and 16 for `Node16`). However, this is not a problem since SIMD instructions can be used to perform the comparison.

Node replacement can become necessary due to (1) the node becoming too full to encompass another insert or due to (2) the node becoming underfull. In both cases, the required steps for replacing a node are the same:

1. Both the node and its parent are locked.
2. A new node of the appropriate type is created and initialized by copying over all entries from the old node.
3. The location within the parent pointing to the old node is changed to the new node using an atomic store.
4. The old node is unlocked and marked as *obsolete*. The parent node is unlocked.

Like with Optimistic Lock Coupling, once a node is not reachable in the most recent version of the tree (after step 3), the node must be marked as obsolete before being unlocked. Any writer waiting for that lock will detect that the node is obsolete and will restart the entire insert or delete operation. For readers, in contrast, it is safe to read from obsolete nodes.

Path compression is illustrated in Figure 4. In the 2-level tree on the left-hand side, the blue node contains the "R" prefix. In order to insert the new key "AS", it is necessary to (1) install a new (green) node and (2) truncate the prefix (at the blue node). Individually, both steps can be performed atomically: Installing the new node can be done by a single (atomic) store. To change the prefix of the node atomically, we store both the prefix and its length in a single 8 byte value, which can be changed atomically[3].

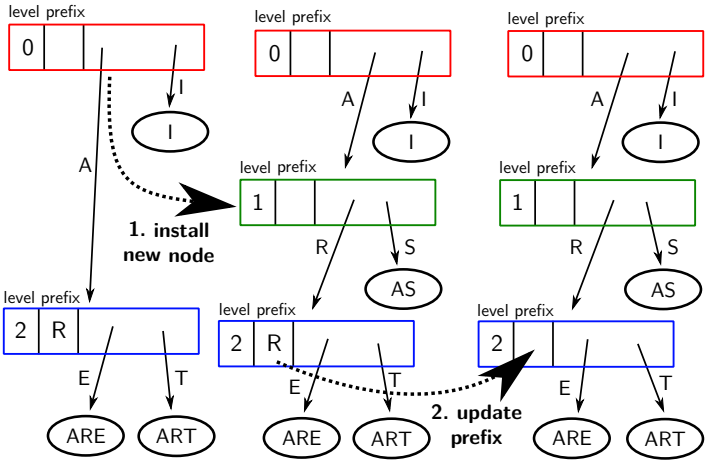

Figure 4: Path compression changes for inserting "AS"

The main difficulty with path compression is that it is not possible to install a new node and truncate the prefix in a single operation. As a consequence, a reader may see the intermediate state in Figure 4. To solve this problem, we augment each node with a `level` field, which stores the height of the node including the prefix and which never changes after a node has been created. With this additional information, the intermediate state in Figure 4 is safe, because a reader will detect that the prefix at the blue node has to be skipped. Similarly, it is also possible that a reader sees the final state of the blue node without having seen the green node before. In that situation, the reader can detect the missing prefix using `level` field and retrieve the missing key from the database.

To summarize, whereas in Optimistic Lock Coupling readers detect changes and restart, with ROWEX it is the responsibility of writers to ensure that reads are safe. Thus ROWEX is not a simple recipe that can be applied to any data structure, but must be carefully adapted to it. In some cases, it might even prove impossible without major changes to the underlying data structure. Nevertheless, we believe that ROWEX is an interesting design point between lock-free techniques and locking.

5. EVALUATION

In this section we experimentally compare a number of ART variants: unmodified ART without support for concurrent modifications, lock coupling with read-write spinlocks, Optimistic Lock Coupling, ROWEX, and hardware transactional memory (HTM) with 20 restarts using a global, elided lock as fallback (as described in [14]). The additional space consumption per node is 4 bytes for lock coupling, 8 bytes for Optimistic Lock Coupling, and 12 bytes for ROWEX. As a competitor we chose Masstree [18], which, to the best of our knowledge, is the fastest publicly available[4], synchronized, order-preserving data structure. Note that the comparison between ART and Masstree is not really "apples-to-apples", as both the synchronization protocol and the data structures themselves differ. All implementations use the `jemalloc` memory allocator and, when required, low-overhead epoch-based memory

[2] The `std::atomic` type does not change the physical layout. Also note that on x86 an atomic load adds very little overhead, as it does not introduce a memory barrier. Making the fields atomic merely prevents the compiler from reordering instructions but does not introduce additional instructions for readers.

[3] In comparison with the original ART implementation this decision reduces the maximum prefix length from 9 to 4. On x86, 16 byte values can also be accessed atomically, which would also allow storing 12 byte prefixes.

[4] https://github.com/kohler/masstree-beta

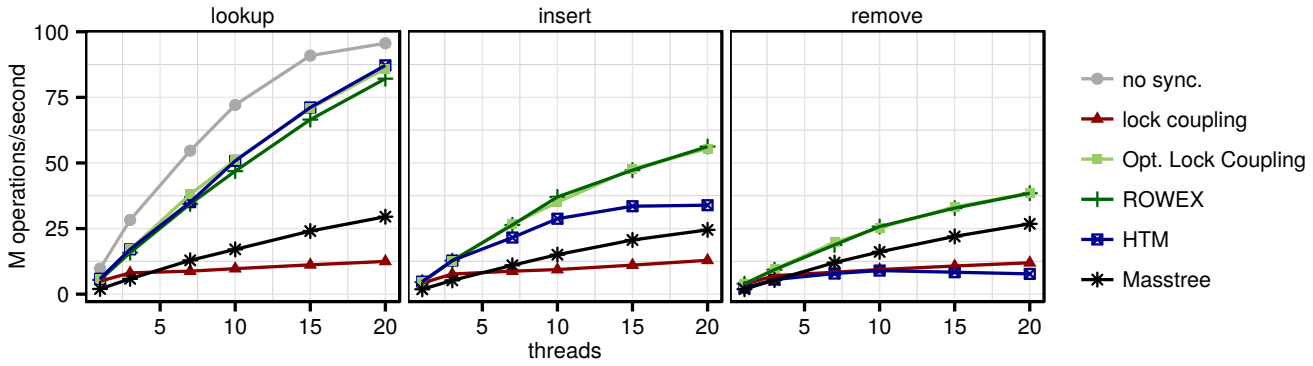

Figure 5: Scalability (50M 8 byte integers)

reclamation. We use a Haswell EP system with an Intel Xeon E5-2687W v3 CPU, which has 10 cores (20 "Hyper-Threads") and 25 MB of L3 cache.

5.1 Scalability

In our first experiment we investigate the scalability using 50M random (sparse) 8-byte integers. This is a low contention workload, because conflicts are unlikely with random keys. Figure 5 shows the results for individually executing lookup, insert, and remove. As expected, the variant without synchronization, which is only shown for the read-only experiment, performs best. Lock coupling does not scale well, even in the lookup experiment and although we use read-write locks. Optimistic Lock Coupling and ROWEX, in contrast, scale very well and have very similar performance, with Optimistic Lock Coupling being slightly (around 7%) faster for lookup. The HTM variant also performs very well on the lookup experiment, but is slightly slower for insert and significantly slower for remove. We verified that the reason for this is (unnecessary) contention in the memory allocator causing frequent aborts. Masstree scales well but with short keys its overall performance is significantly lower than with ART.

To better understand the lookup results in Figure 5, we measured some important CPU statistics:

	1 thread / 20 threads [per lookup]			
	cycles	instruct.	L1 misses	L3 misses
no sync	211 / 381	123 / 124	4.3 / 4.5	1.7 / 1.8
lock coupling	**418 / 2787**	242 / 243	**5.2 / 9.0**	1.8 / 2.0
Opt. Lock Coup.	348 / 418	187 / 187	5.3 / 5.7	1.8 / 2.0
ROWEX	375 / 427	248 / 249	5.6 / 5.8	1.9 / 1.9
HTM	347 / 428	132 / 135	4.3 / 4.5	1.7 / 2.1
Masstree	982 / 1231	897 / 897	20.5 / 21.1	6.5 / 7.1

Single-threaded, the overhead of Optimistic Lock Coupling in comparison with the non-synchronized variant is around 65%, mostly due to additional instructions. With 20 threads, the overhead is reduced to only 10%, likely due to longer delays in the memory controller. With lock coupling, because of cache line invalidations caused by lock acquisitions, the number of L1 misses (highlighted) increases significantly when going from 1 to 20 threads and the CPU cycles increase by a factor of 6.6×. The slowdown would be even larger on multi-socket systems, since such systems do not have a shared cache for inter-thread communication. The CPU statistics also explain why ART is significantly faster than Masstree with integer keys. The Optimistic Lock Coupling variant of ART, for example, requires 4.8× fewer instructions and 3.6× fewer L3 misses than Masstree.

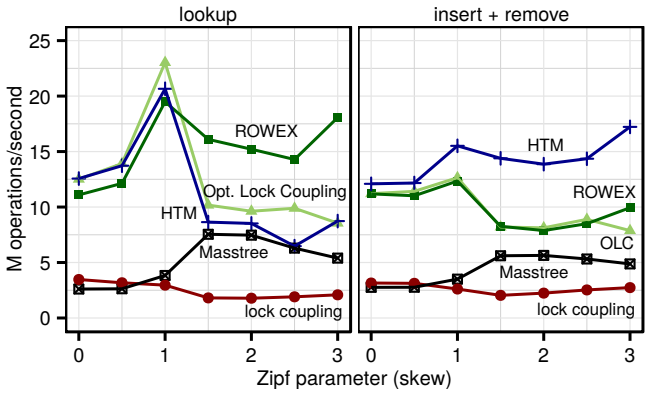

Figure 6: Performance under contention (1 lookup thread and 1 insert+remove thread)

5.2 Contention

In order to show the effect of contention, we measured simultaneous lookup and update operations (insert+remove) in a tree with 10M dense 8 byte integer keys. We used 1 lookup thread and 1 update thread and varied the skewness of the keys going from uniform to extremely skewed (reading and modifying the same key 83% of the time). The lookup results (left-hand side of Figure 6) show that with higher skew most variants initially become faster (due to better cache locality), before eventually slowing down under very high contention. ROWEX is the only exception, as its lookup performance stays very high even under extreme contention due to the non-blocking reads. The performance of the update thread (right-hand side of Figure 6) is generally similar to the lookup performance. One exception is HTM, which has higher performance for the update thread than for the lookup thread, because Intel's transactional memory implementation favors writers over reader [17].

5.3 Strings

Besides integer keys, we also measured the lookup performance for three real-world string data sets. The "genome" data set has 256K strings of average length 10.0, "Wikipedia" contains 16M article titles of average length 22.4, and "URL" has 6.4M URLs of average length 63.3. The lookup and insert performance with 20 threads is shown in Figure 7. Masstree closes its performance gap to ART with longer keys. Again, lock coupling is very slow and the synchronized ART variants are slightly slower than the unsynchronized variant.

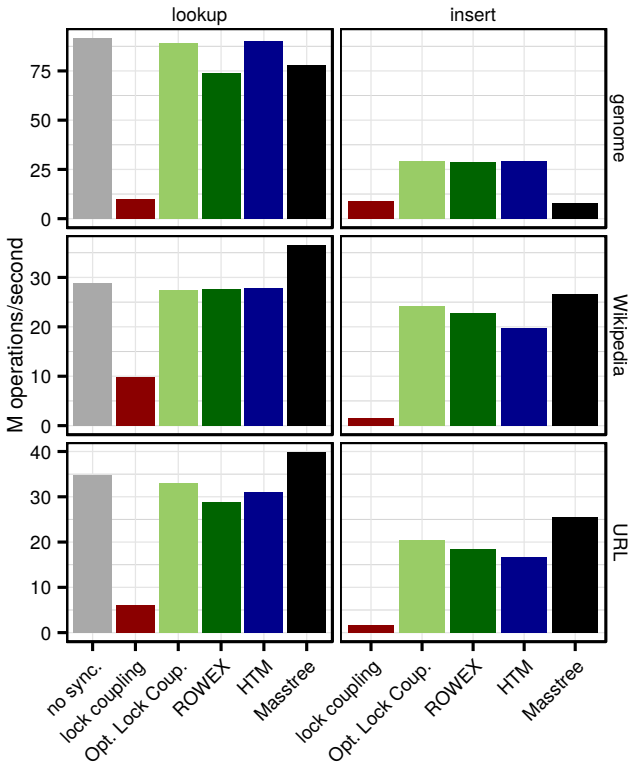

Figure 7: Performance for string data with 20 threads

5.4 Code Complexity

To give a rough indication for the relative complexity of the implementations, we counted the core algorithmic C++ code lines excluding the lock implementations, garbage collection, comments, and empty lines:

	lookup	insert	remove
no synchronization	29	95	87
HTM	30	96	88
lock coupling	41	136	139
Optimistic Lock Coupling	44	148	143
ROWEX	34	200	156

Using HTM is quite trivial, it merely requires wrapping each transaction in a hardware transaction, which we implemented with a `Transaction` object that starts the HTM transaction in the constructor and commits the transaction in the destructor. The two lock coupling variants require more changes, with the optimistic variant being only marginally larger. ROWEX requires most additional code, in particular for the insert and remove operations. The ROWEX numbers actually underestimate the complexity of ROWEX, since its protocol is fairly subtle in comparison with the other variants.

6. RELATED WORK

Concurrency control is one of the major areas in the database field. Most prior work, however, focuses on high-level user transactions, and not so much on low-level synchronization of data structures, which is what we study in this work.

Hand-over-hand locking, i.e., the idea underlying Optimistic Lock Coupling, was used to synchronize binary search trees [2]. ROWEX-like ideas have also been used before, for example by the FOEDUS system [9]. The goal of this paper has been to consolidate these ideas and to present them as general building blocks as opposed to clever tricks used to synchronize individual data structures.

The traditional method of synchronizing B-trees, lock coupling, was proposed by Bayer and Schkolnick [1]. Graefe's surveys [6, 5] on low-level synchronization of B-trees, which summarize decades of accumulate wisdom in the database community, focus on fine-grained locking. Unfortunately, as was already observed by Cha et al. [4] in 2001, lock coupling simply does not scale on modern multi- and many-core CPUs. Since then, the problem has become even more pronounced and the trend is likely to continue. Like Optimistic Lock Coupling, Cha et al.'s solution (OLFIT) uses versions to detect changes within a node. In contrast to Optimistic Lock Coupling, OLFIT does not keep track of versions across multiple nodes, but uses the B-link tree idea [11] to support concurrent structure modification operations.

Both the *Bw-Tree* [15] and *Masstree* [18] are two B-tree proposals published in 2012, and both eschew traditional lock coupling. The Bw-Tree is latch-free and its nodes can be, due to their relatively large size, moved to disk/SSD [16]. To allow concurrent modifications, delta records are pre-pended to a list of existing changes and the (immutable) B-tree node at the end of the list itself. Nodes do not store physical pointers but offsets into a mapping tables that points to the delta lists and allows one to atomically add new delta records. After a number of updates the deltas and the node are consolidated to a new node. The Bw-Tree has been designed with concurrency in mind and its synchronization protocol is highly sophisticated; structure modification operations (e.g., node splits) are very complex operations involving multiple steps.

Masstree [18] is a hybrid B-tree/trie data structure and, like ART, exclusively focuses on the in-memory case. Like the Bw-Tree, Masstree was designed with concurrency in mind, but the designers took different design decisions. The synchronization protocol of Masstree relies on a mix of local locks, clever use of atomic operations, and hand-over-hand locking (the idea underlying Optimistic Lock Coupling). Like Optimistic Lock Coupling, but unlike ROWEX, Masstree lookups must, in some cases, be restarted when a conflict with a structure-modifying operation is detected.

Previous work has shown that Hardware Transactional Memory can be used to synchronize ART [13, 14] and B-trees [7] with very little effort using elided coarse-grained HTM locks. Makreshanski et al. [17] confirmed these findings in an in-depth experimental study but found that performance with HTM can degrade with large tuples or heavy contention. Cervini et al. [3] found that replacing fine-grained locks with (elided) HTM locks at the same granularity does not improve performance. HTM also has the obvious disadvantage of requiring special hardware support, which is not yet widespread.

7. SUMMARY AND DISCUSSION

We presented two synchronization protocols for ART that have good scalability despite relying on locks. The first protocol, *Optimistic Lock Coupling* is very simple, requires few changes to the underlying data structure, and performs very well as long as conflicts are not too frequent. The *Read-Optimized Write EXclusion (ROWEX)* protocol is more complex, but has the advantage that reads never block. ROWEX generally requires changes to the data structure itself, as opposed to simply adding a lock at each node. However, in our experience, while synchronizing an existing, non-trivial data structure using ROWEX may be non-trivial, it is, at least, realistic. Truly lock-free algorithms, in contrast, are much more complicated. They also generally require radical changes and

additional indirections to the underlying data structure. The Bw-Tree, for example, requires an indirection table that causes additional cache misses whenever a node is accessed. Similarly, the state-of-the-art lock-free hash table, the split-ordered list [19], requires "dummy" nodes—again at the price of more cache misses.

It is an open question how a lock-free variant of ART would look like and how well it would perform. We speculate that it would likely be significantly slower than the Optimistic Lock Coupling and ROWEX implementations. We therefore argue that one should not discount the use of locks as long as these locks are infrequently acquired like in our two protocols. Optimistic Lock Coupling and ROWEX are two pragmatic paradigms that result in very good overall performance. We believe both to be highly practical and generally applicable.

8. REFERENCES

[1] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9, 1977.

[2] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPOPP*, pages 257–268, 2010.

[3] D. Cervini, D. Porobic, P. Tözün, and A. Ailamaki. Applying HTM to an OLTP system: No free lunch. In *DaMoN*, 2015.

[4] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, 2001.

[5] G. Graefe. A survey of B-tree locking techniques. *ACM Trans. Database Syst.*, 35(3), 2010.

[6] G. Graefe. Modern B-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[7] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving in-memory database index performance with Intel transactional synchronization extensions. In *HPCA*, 2014.

[8] A. Kemper, T. Neumann, J. Finis, F. Funke, V. Leis, H. Mühe, T. Mühlbauer, and W. Rödiger. Transaction processing in the hybrid OLTP & OLAP main-memory database system HyPer. *IEEE Data Eng. Bull.*, 36(2):41–47, 2013.

[9] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD*, pages 691–706, 2015.

[10] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. KISS-Tree: Smart latch-free in-memory indexing on modern architectures. In *DaMoN*, 2012.

[11] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[12] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*, 2013.

[13] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *ICDE*, 2014.

[14] V. Leis, A. Kemper, and T. Neumann. Scaling HTM-supported database transactions to many cores. *IEEE Trans. Knowl. Data Eng.*, 28(2), 2016.

[15] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-tree: A B-tree for new hardware platforms. In *ICDE*, 2013.

[16] J. J. Levandoski, D. B. Lomet, and S. Sengupta. LLAMA: a cache/storage subsystem for modern hardware. *PVLDB*, 6(10):877–888, 2013.

[17] D. Makreshanski, J. J. Levandoski, and R. Stutsman. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *PVLDB*, 8(11), 2015.

[18] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[19] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, 2006.

APPENDIX

A. Implementation of Optimistic Locks

The following pseudo code implements optimistic locks. Each node header stores a 64 bit `version` field that is read and written atomically. The two least significant bits indicate if the node is obsolete or if the node is locked, respectively. The remaining bits store the update counter.

```
struct Node
   atomic<uint64_t> version
   ...

uint64_t readLockOrRestart(Node node)
   uint64_t version = awaitNodeUnlocked(node)
   if isObsolete(version)
      restart()
   return version

void checkOrRestart(Node node, uint64_t version)
   readUnlockOrRestart(node, version)

void readUnlockOrRestart(Node node, uint64_t version)
   if version != node.version.load()
      restart()

void readUnlockOrRestart(Node node, uint64_t version,
                         Node lockedNode)
   if version != node.version.load()
      writeUnlock(lockedNode)
      restart()

void upgradeToWriteLockOrRestart(Node node, uint64_t version)
   if !node.version.CAS(version, setLockedBit(version))
      restart()

void upgradeToWriteLockOrRestart(Node node, uint64_t version,
                                 Node lockedNode)
   if !node.version.CAS(version, setLockedBit(version))
      writeUnlock(lockedNode)
      restart()

void writeLockOrRestart(Node node)
   uint64_t version
   do
      version = readLockOrRestart(node)
   while !upgradeToWriteLockOrRestart(node, version)

void writeUnlock(Node node)
   // reset locked bit and overflow into version
   node.version.fetch_add(2)

void writeUnlockObsolete(Node node)
   // set obsolete, reset locked, overflow into version
   node.version.fetch_add(3)

// Helper functions
uint64_t awaitNodeUnlocked(Node node)
   uint64_t version = node.version.load()
   while (version & 2) == 2 // spinlock
      pause()
      version = node.version.load()
   return version

uint64_t setLockedBit(uint64_t version)
   return version + 2

bool isObsolete(uint64_t version)
   return (version & 1) == 1
```

B. Insert with Optimistic Lock Coupling

The following pseudo code implements the insert operation using optimistic lock coupling. Initially, the traversal proceed like in the lookup case without acquiring write locks. Once the node that needs to be modified is found, it is locked. In cases where the node must grow the parent is also locked.

```
insertOpt(key, value, node, level, parent, parentVersion)
   version = readLockOrRestart(node)
   if !prefixMatches(node, key, level)
      upgradeToWriteLockOrRestart(parent, parentVersion)
      upgradeToWriteLockOrRestart(node, version, parent)
      insertSplitPrefix(key, value, node, level, parent)
      writeUnlock(node)
      writeUnlock(parent)
      return
   nextNode = node.findChild(key[level])
   checkOrRestart(node, version)
   if nextNode == null
      if node.isFull()
         upgradeToWriteLockOrRestart(parent, parentVersion)
         upgradeToWriteLockOrRestart(node, version, parent)
         insertAndGrow(key, value, node, parent)
         writeUnlockObsolete(node)
         writeUnlock(parent)
      else
         upgradeToWriteLockOrRestart(node, version)
         readUnlockOrRestart(parent, parentVersion, node)
         node.insert(key, value)
         writeUnlock(node)
      return
   if parent != null
       readUnlockOrRestart(parent, parentVersion)
   if isLeaf(nextNode)
       upgradeToWriteLockOrRestart(node, version)
       insertExpandLeaf(key, value, nextNode, node, parent)
       writeUnlock(node)
       return
   // recurse to next level
   insertOpt(key, value, nextNode, level+1, node, version)
   return
```