Synchronization on Multi-Core CPUs

Viktor Leis

Technische Universität München
Introduction

- this course focuses on high-level concurrency control protocols (logical transaction isolation)
- any implementation of these protocols must also deal with low-level synchronization (thread safety)
- many data structures must thread-safe: index structures, tuple storage, job queues, buffer management data structures, etc.
- low-level synchronization often decides how well a program scales on multi-core CPUs
Case Study: The Adaptive Radix Tree (ART)

- order-preserving index for in-memory database systems
- originally designed for high single-threaded performance
Lock Coupling

- very easy to apply to ART
- modifications only change 1 node and (sometimes) its parent
- can use read/write locks to allow for more concurrency
Lock Coupling

- very easy to apply to ART
- modifications only change 1 node and (sometimes) its parent
- can use read/write locks to allow for more concurrency

```python
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)
```
Performance of Lock Coupling

![Graph showing performance of Lock Coupling. The x-axis represents the number of threads, ranging from 5 to 20. The y-axis represents M operations/second, ranging from 0 to 100. Two lines are shown: one for 'no synchronization' and one for 'lock coupling'. The 'no synchronization' line shows an upward trend as the number of threads increases, while the 'lock coupling' line remains relatively flat.]
Lock-free ART?

- non-blocking data structures are extremely difficult to design, to implement, and to debug
- every non-trivial lock-free data structure is a research contribution
- non-blocking data structures add significant overhead
  - Bw-tree: extra delta records in front of each node
  - Split-ordered list (state-of-the-art lock-free hash table): dummy nodes
- a hypothetical lock-free ART variant would
  - require significant changes to the data structure (path compression is a major issue)
  - likely be slower than the methods presented in the following
scalability

ease of use

fine-grained locking (lock coupling)

lock-free
Hardware Transactional Memory

- Intel’s Haswell microarchitecture introduced hardware support in mainstream CPUs
- (only guarantees in-memory atomicity and isolation, but not durability)
Interface to HTM: Intel Transactional Synchronization Extensions (TSX)

- Restricted Transactional Memory (RTM):
  - XBEGIN: begin
  - XEND: commit
  - XABORT: rollback

- Hardware Lock Elision (HLE):
  - XACQUIRE prefix: “acquire” lock speculatively
  - XRELEASE prefix: release lock speculatively
  - prefix is ignored on older CPUs
Hardware Lock Elision

- elide lock on first try optimistically
- start HTM transaction instead
- if a conflict happens, the lock is actually acquired
How Does the CPU implement HTM?

- local L1 cache (32KB) serves as a buffer for transactional writes and for tracking transactional reads at cache line granularity (64 bytes)
- cache coherency protocol is used to detect conflicts
Limitations of Haswell’s HTM

- size (32KB) and associativity (8-way) of L1 cache limit transaction size
- interrupts, context switches limit transaction duration
- certain (rarely used) instructions always cause abort
Hardware Transactional Memory (HTM)

+ very easy to use (with coarse-grained, elided locks)
+ often scales well
  - requires special (not yet widespread) hardware support
  - sometimes hard to predict/debug behavior
scalability

lock-free

HTM

ease of use

fine-grained locking (lock coupling)
Optimistic Lock Coupling (1)

- add lock and version to each node
- write:
  - acquire lock (exclude other writers)
  - increment version when unlocking
  - do not acquire locks for nodes that are not modified (traverse like a reader)
- read:
  - do not acquire locks, proceed optimistically
  - detect concurrent modifications through versions (and restart if necessary)
  - can track changes across multiple nodes (lock coupling)
Optimistic Lock Coupling (2)

**traditional**

1. lock node A
2. search node A
3. lock node B
4. unlock node A
5. search node B
6. lock node C
7. unlock node B
8. search node C
9. unlock node B

**optimistic**

1. lock node A
2. search node A
3. lock node B
4. unlock node A
5. search node B
6. lock node C
7. unlock node B
8. search node C
9. unlock node B
Optimistic Lock Coupling (2)

Traditional

1. lock node A
2. search node A

Optimistic

3. lock node B
4. unlock node A
5. search node B

6. lock node C
7. unlock node B
8. search node C
9. unlock node B
Optimistic Lock Coupling (2)

**traditional**

1. lock node A
2. search node A
3. lock node B
4. unlock node A
5. search node B
6. lock node C
7. unlock node B
8. search node C
9. unlock node B

**optimistic**

1. read version v3
2. search node A

A

B

C
Optimistic Lock Coupling (2)

**traditional**

1. lock node A
2. search node A
3. lock node B
4. unlock node A
5. search node B
6. lock node C
7. unlock node B
8. search node C
9. unlock node B

**optimistic**

1. read version v3
2. search node A
3. read version v7
4. re-check version v3
5. search node B
6. lock node C
7. unlock node B
8. search node C
9. unlock node B
Optimistic Lock Coupling (2)

traditional

1. lock node A
2. search node A
3. lock node B
4. unlock node A
5. search node B
6. lock node C
7. unlock node B
8. search node C
9. unlock node B

optimistic

1. read version v3
2. search node A
3. read version v7
4. re-check version v3
5. search node B
6. re-check version v7
7. search node C
8. re-check version v5
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)

lookupOpt(key, node, level, parent, version)
    version = readLockOrRestart(node)
    if parent != null
        readUnlockOrRestart(parent, version)
    // check if prefix matches, may increment level
    if !prefixMatches(node, key, level)
        readUnlockOrRestart(node, version)
        return null // key not found
    // find child
    nextNode = node.findChild(key[level])
    checkOrRestart(node, version)
    if isLeaf(nextNode)
        value = getLeafValue(nextNode)
        readUnlockOrRestart(node, version)
        return value // key found
    if nextNode == null
        readUnlockOrRestart(node, version)
        return null // key not found
    // recurse to next level
    return lookupOpt(key, nextNode, level+1, node, version)
Optimistic Lock Coupling (4)

+ can easily be applied to most data structures (no modifications necessary)
+ scales well
+ low overhead
- can lead to (unnecessary) aborts
scalability

- lock-free

- optimistic lock coupling

- fine-grained locking (lock coupling)

ease of use

- HTM
Read-Optimized Write EXclusion (ROWEX) (1)

- add lock to each node
- write:
  - acquire lock (excludes writers)
  - make sure than any modification leaves the tree in a state safe for readers
- read:
  - simply proceed without observing locks or versions
Read-Optimized Write EXclusion (ROWEX) (2)

+ scales well
+ reads are non-blocking (always successful and there are no restarts)
+ easier to implement than lock-free data structures
- more difficult to implement than Optimistic Lock Coupling (requires modifications to the underlying data structure)
Synchronizing ART with ROWEX

- **local modifications:**
  - make key and child pointer accesses atomic (`std::atomic`)
  - make Node4 and Node16 unsorted and append-only

- **grow/shrink a node:**
  - lock node and its parent
  - create new node and copy entries
  - set parent pointer to the new node

- **path compression:**
  - modify prefix atomically
  - add `level` field to each node
Lookup (50M 8B Integers)
Insert (50M 8B Integers)

![Graph showing performance comparison of different lock coupling algorithms: lock coupling, Opt. Lock Coupling, ROWEX, HTM, and Masstree. The graph plots M operations/second against the number of threads.](image)
Lookup/Insert/Remove the Same Key (High Contention, 2 threads)

The diagrams show the performance of different data structures and optimization techniques across varying Zipf parameters (skew). For both lookup and insert + remove operations, the y-axis represents the number of M operations per second, and the x-axis represents the Zipf parameter (skew) ranging from 0 to 3.

- **Lookup**:
  - ROWEX
  - Opt. Lock Coupling
  - Masstree
  - HTM

- **Insert + Remove**:
  - ROWEX
  - HTM
  - OLC

Each graph plots the performance improvement as the Zipf parameter increases, demonstrating how different techniques handle contention and scalability in high-concurrency scenarios.
Summary

- traditional fine-grained locking does not scale for tree-like index structures
- locks are fine if they are only acquired by writers and only on nodes that are modified
- Optimistic Lock Coupling is a highly practical alternative to the lock-free paradigm

Source: https://github.com/flode/ARTSynchronized