Synchronizing Data Structures

Overview

- caches and atomics
- list-based set
- memory reclamation
- Adaptive Radix Tree
- B-tree
- Bw-tree
- split-ordered list
- hardware transactional memory

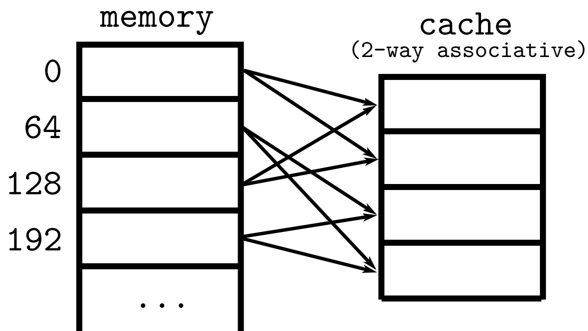
Caches

modern CPUs consist of multiple CPU cores and

- per-core registers
- per-core write buffers
- per-core caches (L1, L2)
- shared cache (L3)
- shared main memory

Cache Organization

- caches are organized in fixed-size chunks called *cache lines*
- on Intel CPUs a cache line is 64 bytes
- data accesses go through cache, which is transparently managed by the CPUs
- caches implement a replacement strategy to evict pages
- associativity: how many possible cache locations does each memory location have?



Cache Coherency Protocol

- although cores have private caches, the CPU tries to hide this fact
- CPU manages caches and provides the illusion of a single main memory using a *cache coherency protocol*
- example: MESI protocol, which has the following states:
  - *Modified*: cache line is only in current cache and has been modified
  - *Exclusive*: cache line is only in current cache and has not been modified
  - *Shared*: cache line is in multiple caches
  - *Invalid*: cache line is unused
- Intel uses the MESIF protocol, with an additional *Forward* state
- *Forward* is a special *Shared* state indicating a designated "responder"

Optimizations

- both compilers and CPUs reorder instructions, eliminate code, keep data in register, etc.
- these optimizations are sometimes crucial for performance
- for single-threaded execution, compilers and CPUs guarantee that the semantics of the program is unaffected by these optimizations (as if executed in program order)
- with multiple threads, however, a thread may observe these “side effects”
- in order to write correct multi-threaded programs, synchronization primitives must be used

Example

```
int global(0);

void thread1() {
   while (true) {
      while (global%2 == 1); // wait
      printf("ping\n");
      global++;
   }
}

void thread2() {
   while (true) {
      while (global%2 == 0); // wait
      printf("pong\n");
      global++;
   }
}
```

C++11 Memory Model

- accessing a shared variable by multiple threads where at least thread is a writer is a race condition
- according to the C++11 standard, race conditions are *undefined behavior*
- depending on the compiler and optimization level, undefined behavior may cause *any* result/outcome
- to avoid undefined behavior when accessing shared data one has to use the `std::atomic` type[1]
- atomics provide atomic load/stores (no torn writes), and well-defined ordering semantics

[1] `std::atomic` is similar to Java's `volatile` keyword but different from C++'s `volatile`

Atomic Operations in C++11

- compare-and-swap (CAS): `bool std::atomic_compare_exchange_strong(T& expected, T desired)`
- there is also a weak CAS variant that may fail even if `expected` equals `desired`, on x86-64 both variants generate the same code
- exchange: `std::exchange(T desired)`
- arithmetic: addition, subtraction
- logical: and/or/xor

Naive Spinlock (Exchange)

```
struct NaiveSpinlock {
    std::atomic<int> data;

    NaiveSpinlock() : data(0) {}

    void lock() {
        while (data.exchange(1)==1);
    }

    void unlock() {
        data.store(0); // same as data = 0
    }
};
```

Naive Spinlock (CAS)

```
struct NaiveSpinlock {
    std::atomic<int> data;

    NaiveSpinlock() : data(0) {}

    void lock() {
        int expected;
        do {
            expected = 0;
        } while (!data.compare_exchange_strong(expected, 1));
    }

    void unlock() {
        data.store(0); // same as data = 0
    }
};
```

Sequential Consistency and Beyond

- by default, operations on `std::atomic` types guarantee *sequential consistency*
- non-atomic loads and stores are not reordered around atomics
- this is often what you want
- all `std::atomic` operations take one or two optional `memory_order` parameter(s)
- allows one to provide less strong guarantees (but potentially higher performance), the most useful ones on x86-64 are:
  - `std::memory_order::memory_order_seq_cst`:
    sequentially consistent (the default)
  - `std::memory_order::memory_order_release` (for stores):
    may move non-atomic operations before the store (i.e., the visibility of the store can be delayed)
  - `std::memory_order::memory_order_relaxed`:
    guarantees atomicity but no ordering guarantees[2]
- nice tutorial: `https://assets.bitbashing.io/papers/lockless.pdf`

[2] sometimes useful for data structures that have been built concurrently but are later immutable

Spinlock

```
struct Spinlock {
   std::atomic<int> data;
   Spinlock() : data(0) {}

   void lock() {
      for (unsigned k = 0; !try_lock(); ++k)
         yield(k);
   }

   bool try_lock() {
      int expected = 0;
      return data.compare_exchange_strong(expected, 1);
    }

   void unlock() {
      data.store(0, std::memory_order::memory_order_release);
   }

   void yield();
};
```

Yielding

```
// adapted from Boost library
void Spinlock::yield(unsigned k) {
   if (k < 4) {
   } else if (k < 16) {
      _mm_pause();
   } else if ((k < 32) || (k & 1)) {
      sched_yield();
   } else {
      struct timespec rqtp = { 0, 0 };
      rqtp.tv_sec = 0;
      rqtp.tv_nsec = 1000;
      nanosleep(&rqtp, 0);
   }
}
```

Lock Flavors

- there are many different lock implementations
- C++: `std::mutex`, `std::recursive_mutex`
- pthreads: `pthread_mutex_t`, `pthread_rwlock_t`
- on Linux blocking locks are based on the `futex` system call
- `https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/Mutex_Flavors.html`:

TBB type	Scalable	Fair	Recursive	Long Wait	Size
mutex	OS dependent	OS dependent	no	blocks	$\geq$ 3 words
recursive_mutex	OS dependent	OS dependent	yes	blocks	$\geq$ 3 words
spin_mutex	no	no	no	yields	1 byte
speculative_spin_mutex	HW dependent	no	no	yields	2 cache lines
queuing_mutex	yes	yes	no	yields	1 word
spin_rw_mutex	no	no	no	yields	1 word
speculative_spin_rw_mutex	HW dependent	no	no	yields	3 cache lines
queuing_rw_mutex	yes	yes	no	yields	1 word
null_mutex	moot	yes	yes	never	empty
null_rw_mutex	moot	yes	yes	never	empty

Atomics on x86-64

- atomic operations only work on 1, 2, 4, 8, or 16 byte data that is aligned
- atomic operations use `lock` instruction prefix
- CAS: `lock cmpxchg`
- exchange: `xchg` (always implicitly locked)
- read-modify-write: `lock add`
- memory order can be controlled using fences (also known as barriers): `_mm_lfence()`, `_mm_sfence()`, `_mm_mfence()`
- locked instructions imply full barrier
- fences are very hard to use, but atomics generally make this unnecessary

x86-64 Memory Model

- x86-64 implements Total Store Order (TSO), which is a strong memory model
- this means that x86 mostly executes the machine code as given
- loads are not reordered with respect to other loads, writes are not reordered with respect to other writes
- however, writes are buffered (in order to hide the L1 write latency), and *reads are allowed to bypass writes*
- a fence or a locked write operations will flush the write buffer (but will not flush the cache)
- important benefit from TSO: sequentially consistent loads do not require fences

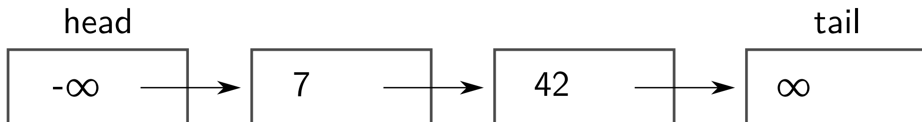
Weakly-Ordered Hardware

- many microarchitectures (e.g., ARM) are weakly-ordered
- on the one hand, on such systems many explicit fences are necessary
- on the other hand, the CPU has more freedom to reorder
- ARMv8 implements acquire/release semantics in hardware (`lda` and `str` instructions)
- `https://en.wikipedia.org/wiki/Memory_ordering`:

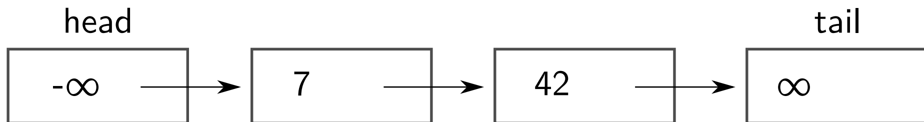
	Alpha	ARM	IBM		SPARC			Intel		
		v7	POWER	zArch	RMO	PSO	TSO	x86	x86-64	IA-64
Loads reord. after loads	Y	Y	Y		Y					Y
Loads reord. after stores	Y	Y	Y		Y					Y
Stores reord. after stores	Y	Y	Y		Y	Y				Y
Stores reord. after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reord. with loads	Y	Y	Y		Y					Y
Atomic reord. with stores	Y	Y	Y		Y	Y				Y
Dependent loads reord.	Y									
Incoh. instr. cache pipel.	Y	Y	Y		Y	Y	Y	Y		Y

Concurrent List-Based Set

- operations: insert(key), remove(key), contains(key)
- keys are stored in a (single-)linked list sorted by key
- `head` and `tail` are always there (“sentinel” elements)



Why CAS Is Not Enough



- thread A: remove(7)
- thread B: insert(9)

Coarse-Grained Locking

- use a single lock to protect the entire data structure

+ very easy to implement

− does not scale at all

Lock Coupling

- also called “hand-over-hand locking” or “crabbing”
- hold at most two locks at a time
- interleave lock acquisitions/release pair-wise
- may use read/write locks to allow for concurrent readers

+ easy to implement
+ no restarts
− does not scale

Optimistic

- "trust, but verify"
- traverse list optimistically without taking any locks
- lock 2 nodes (`predecessor` and `current`)
- validate: traverse list again and check that `predecessor` is still reachable and points to `current`
- if validation fails, unlock and restart

+ lock contention unlikely
− must traverse list twice
− readers acquire locks

Optimistic Lock Coupling

- general technique that can be applied to many data structures (e.g., ART, B-tree)
- associate lock with update counter
- write:
  - acquire lock (exclude other writers)
  - increment counter 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 counters (and restart if necessary)
  - can track changes across multiple nodes (lock coupling)

\+ easy to implement

\+ scalable

− restarts

Non-Blocking Algorithms

- killing a thread at any point of time should not affect consistency of the data structure (this precludes locks)
- non-blocking data structures may be beneficial for (soft) real-time applications
- classification:
  - wait-free: every operation is guaranteed to succeed (in a constant number of steps)
  - lock-free: overall progress is guaranteed (some operations succeed, while others may not finish)
  - obstruction-free: progress is only guaranteed if there is no interference from other threads

Read-Optimized Write Exclusion (Lazy)

- contains is wait-free
- add/remove traverse list only once (as long as there is no contention)
- add `marker` to each node for *logically* deleting a key
- invariant: every unmarked node is reachable
- contains: no need to validate; if a key is not found or is marked, the key is not in the set
- add/remove:
  1. lock `predecessor` and `current`
  2. check that both are unmarked and that `predecessor` points to `current`
  3. remove marks first, then updates next pointer of `predecessor`

\+ no restarts

\+ scalable

– insert/remove lock

Lock-Free List

- insert and remove are lock-free, contains is wait-free
- remove: marks node for deletion, but does not physically remove it
- marker is stored within the `next` pointer (by stealing a bit of the pointer)
- insert and remove:
  - do not traverse marked node, but physically remove it during traversal using CAS
  - if this CAS fails, restart from `head`
- contains traverses marked nodes (but needs same check as Lazy variant)

+ contains always succeeds
+ scalable
− insert/remove may restart

Synchronization Paradigms

	complexity	scalability	overhead
Coarse-Grained Locking	+	--	+
Lock Coupling	+	-	~
Optimistic	-	+	-
Optimistic Lock Coupling	+	+	+
ROWEX	-	+	+
Lock-Free	--	++	--

Memory Reclamation for Lock-Free Data Structures

- after deleting a node in a lock-free data structure, readers might still be accessing that node
- freeing/reusing that node immediately can cause correctness problems and/or crashes
- one must not physically delete a node unless it is ensured that no threads can access that node
- how long should one wait until the node is physically deleted?
- garbage-collected languages usually do not have this particular problem

Reference Counting

- associate every node with an atomic counter
- effectively results in similar behavior as locks

+ easy to use
– does not scale

Hazard Pointers

- observation: most lock-free operations only require a bounded number of node pointers at any point in time
- during traversal, one can store these *hazard* pointers into thread-local locations
- before physically removing a node, check if any thread has a hazard pointer referencing that node

+ non-blocking

– high overhead due to required fences

– error-prone (requires invasive changes to data structure)

Epoch-Based Memory Reclamation

- global epoch counter (incremented infrequently)
- per-thread, local epoch counters
- before every operation: enter epoch by setting local epoch to global epoch
- after every operation: set local epoch to ∞ indicating that this thread is not accessing the data structure
- tag nodes to be deleted with current global epoch
- defer physically deleting nodes
- it is safe deleting nodes are older than the minimum of all local epochs

\+ low overhead

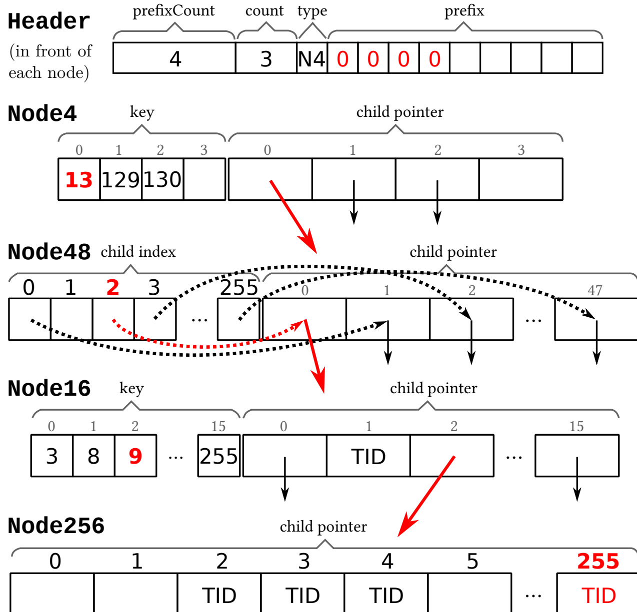
\+ no need to change data structure code

− a single slow/blocking thread may prevent *any* memory reclamation

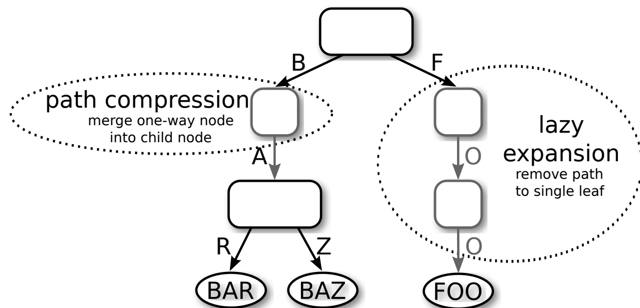
ABA Problem

- a compare-and-swap on a pointer structure may succeed even though the pointer has been changed in the mean time (from A to B back to A)
- this is a correctness issue for some lock-free data structures (e.g., queues)
- whether this problem occurs depends on data structure and the memory reclamation strategy

Adaptive Radix Tree (ART)



Path Compression and Lazy Expansion



Lock Coupling

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

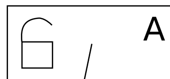
Optimistic Lock Coupling

- add lock and version to each node
- how to detect that root node changed?
  1. extra optimistic lock for root pointer (outside of any node)
  2. always keep the same Node256 as root node (similar to static head element in list-based set)
  3. after reading the version of the current root, validate that the root pointer still points to this node

Optimistic Lock Coupling (2)

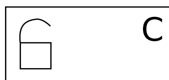
traditional | **optimistic**

1. lock node A
2. search node A



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

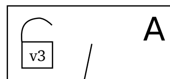
6. lock node C
7. unlock node B
8. search node C
9. unlock node B



Optimistic Lock Coupling (2)

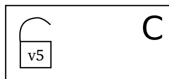
traditional **optimistic**

1. lock node A
2. search node A



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

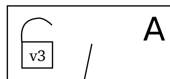
6. lock node C
7. unlock node B
8. search node C
9. unlock node B



Optimistic Lock Coupling (2)

traditional | **optimistic**

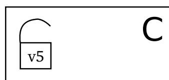
1. lock node A
2. search node A



1. read version v3
2. search node A

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

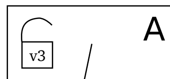
6. lock node C
7. unlock node B
8. search node C
9. unlock node B



Optimistic Lock Coupling (2)

traditional | **optimistic**

1. lock node A
2. search node A

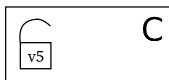


1. read version v3
2. search node A

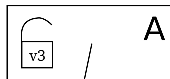
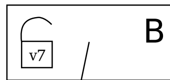
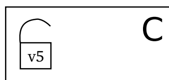
3. lock node B
4. unlock node A
5. search node B

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		optimistic
1. lock node A 2. search node A		1. read version v3 2. search node A
3. lock node B 4. unlock node A 5. search node B		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		6. read version v5 7. re-check version v7 8. search node C 9. re-check version v5

Lock Coupling Vs. Optimistic Lock Coupling

```
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)
```

Insert using Optimistic Lock Coupling

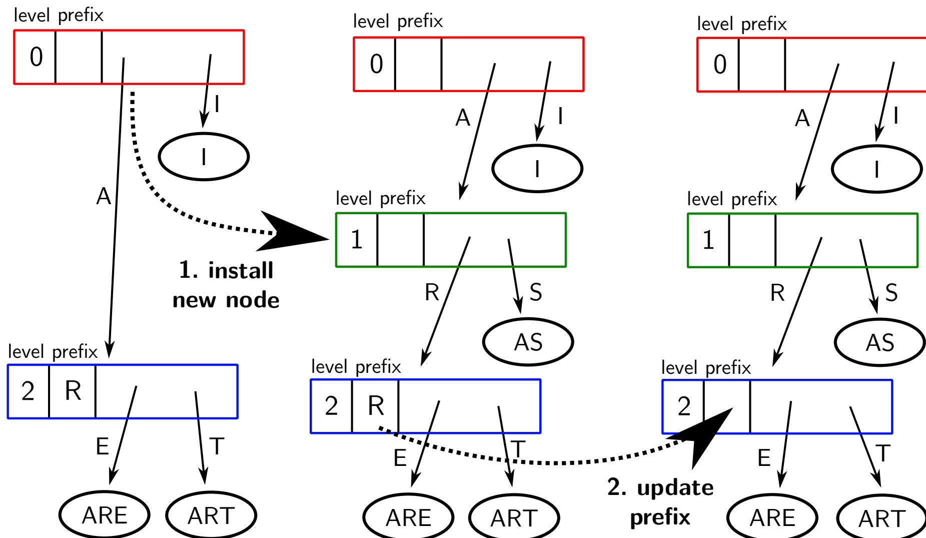
```
1  insertOpt(key, value, node, level, parent, parentVersion)
2     version = readLockOrRestart(node)
3     if !prefixMatches(node, key, level)
4        upgradeToWriteLockOrRestart(parent, parentVersion)
5        upgradeToWriteLockOrRestart(node, version, parent)
6        insertSplitPrefix(key, value, node, level, parent)
7        writeUnlock(node)
8        writeUnlock(parent)
9        return
10    nextNode = node.findChild(key[level])
11    checkOrRestart(node, version)
12    if nextNode == null
13       if node.isFull()
14          upgradeToWriteLockOrRestart(parent, parentVersion)
15          upgradeToWriteLockOrRestart(node, version, parent)
16          insertAndGrow(key, value, node, parent)
17          writeUnlockObsolete(node)
18          writeUnlock(parent)
19       else
20          upgradeToWriteLockOrRestart(node, version)
21          readUnlockOrRestart(parent, parentVersion, node)
22          node.insert(key, value)
23          writeUnlock(node)
24       return
25    if parent != null
26        readUnlockOrRestart(parent, parentVersion)
27    if isLeaf(nextNode)
28        upgradeToWriteLockOrRestart(node, version)
29        insertExpandLeaf(key, value, nextNode, node, parent)
30        writeUnlock(node)
31        return
32    // recurse to next level
33    insertOpt(key, value, nextNode, level+1, node, version)
34    return
```

ART with ROWEX

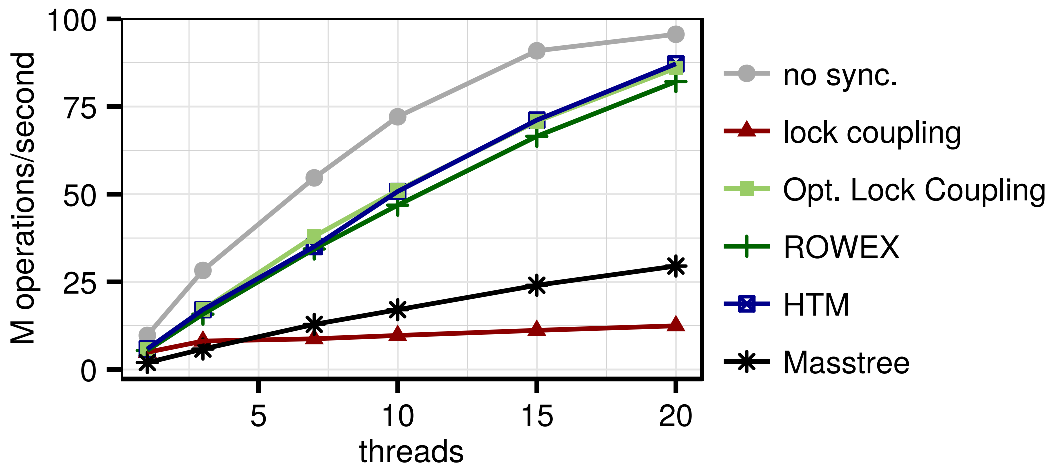
- local modifications:
  - make key and child pointers `std::atomic` (for readers)
  - make `Node4` and `Node16` become unsorted and append-only
- grow/shrink a node:
  1. lock node and its parent
  2. create new node and copy entries
  3. set parent pointer to the new node
  4. mark old node as obsolete
  5. unlock node and parent
- path compression:
  - modify 16-byte prefix atomically (4-byte length, 12-byte prefix)
  - add `level` field to each node
- much more complicated than OLC, requires invasive changes to data structure

Path Compression with ROWEX

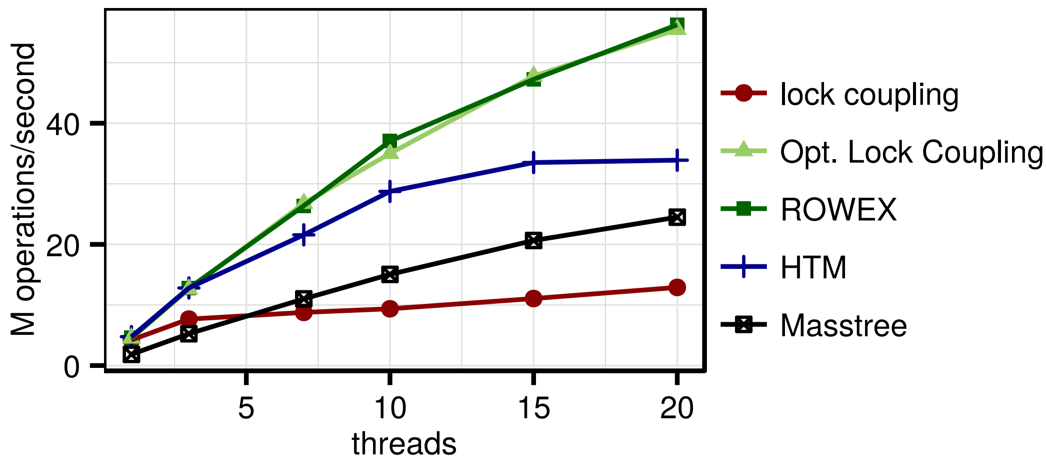
- insert("AS"):



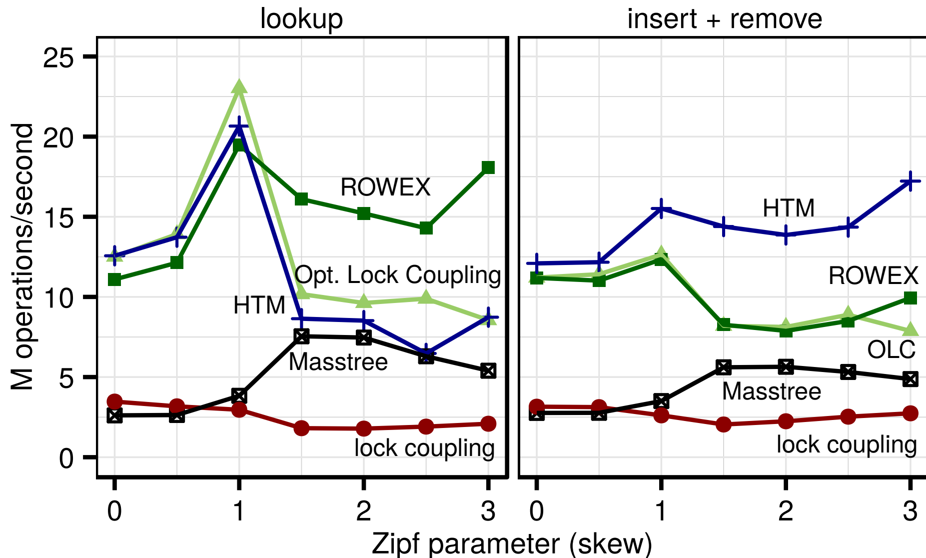
Lookup (50M 8B Integers)



Insert (50M 8B Integers)



Contention



Synchronizing B-trees

- we assume B+-trees (inner nodes only store pointers, not values)
- for insert/delete there are two cases:
  1. single-node change (normal, frequent case)
  2. structure modification operation (during split/merge, infrequent)

Lock Coupling

- must detect root node changes (e.g., additional lock)
- structure modification operations may propagate up multiple levels:
  - eagerly split full inner nodes during traversal (good solution for fixed-size keys)
  - restart operation from the root holding all locks

Optimistic Lock Coupling

- optimistic lock coupling can be applied (after solving the problem discussed on the previous slide)
- writers usually only lock one leaf node (very important for scalability)
- additional issues due to optimism:
  - infinite loops: one has to ensure that the intra-node (binary) search terminates in the presence of concurrent modifications
  - segmentation faults/invalid behavior: a pointer read from a node may be invalid (additional check needed)
- OLC is a good technique for B-trees

B-Link Tree

- B-tree synchronization with only a single lock at a time (no coupling)
- observation: as long as there are only splits (no merges), the key that is being searched may have moved to a right neighbor
- solution: add *next* pointers to inner and leaf nodes, operations may have to check neighbor(s)
- fence keys may help determining whether it is necessary to traverse to neighbor
- the B-Link tree idea was very important when data was stored on disk but is also sometimes used for in-memory B-tree synchronization (e.g., OLFIT)

B-tree ROWEX?

- use B-Link tree idea to enable readers to handle concurrent splits
- lock-less per-node operations can be implemented using a design similar to slotted pages
- not possible to keep keys sorted (must scan all keys)
- not clear whether this is a useful design

Lock-Free: Bw-tree

- slides courtesy of Andy Pavlo

Transactional Memory

- implementing efficient synchronization protocols is very difficult and error-prone
- transactions are a very easy way to manage concurrency, they provide atomicity, isolation, and concurrency
- it would be nice to have transactions directly in the programming language
- software transactional memory (STM): a language extension and/or software library implementing transactions (large research field, often significant overhead)
- hardware transactional memory (HTM): the CPU implements transactions in hardware
- HTM was invented by Herlihy and Moss in 1993 (“Transactional Memory: Architectural Support for Lock-Free Data Structures”)

Hardware Transactional Memory Implementations

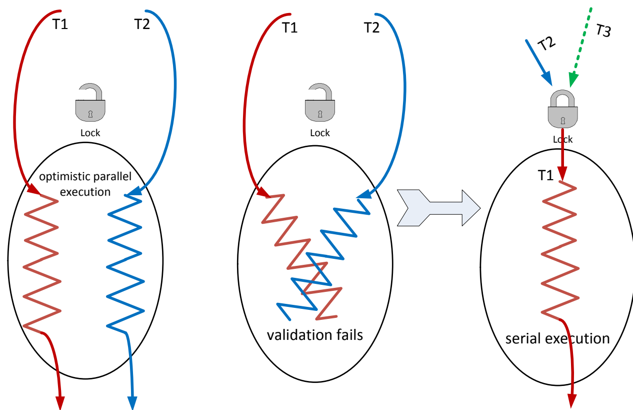
- Intel:
  - introduced by Haswell microarchitecture (2013)
  - disabled in firmware update due to (obscure) hardware bug
  - enabled on Skylake
- IBM:
  - Power8
  - Blue Gene/Q (supercomputers)
  - System z (mainframes)
- each of these implementations has different characteristics and a different ISA interface
- AMD does not implement HTM yet

Intel Transactional Synchronization Extensions (TSX)

- Hardware Lock Elision (HLE): interface looks like mutex-based code, can be used HTM to speed up existing locking code
- Restricted Transactional Memory (RTM): explicit transaction instructions
- on success/commit, both approaches make all memory changes visible atomically
- on abort, both approaches undo all register and memory changes

Hardware Lock Elision (1)

- elide lock on first try optimistically
- start HTM transaction instead
- if a conflict happens, the lock is actually acquired and the transactions is restarted



Using Hardware Lock Elision

- exposed as special instruction prefixes (`xacquire` and `xrelease`), which are annotations to load/store instructions
- prefixes are ignored on older CPUs: code still works on older CPUs and always acquires lock

Hardware Lock Elision Example

```
struct SpinlockHLE {
   int data;
   SpinlockHLE() : data(0) {}

   void lock() {
      asm volatile("1:  movl $1, %%eax                 \n\t"
                   "    xacquire lock xchgl %%eax, (%0) \n\t"
                   "    cmpl $0, %%eax                 \n\t"
                   "    jz 3f                          \n\t"
                   "2:  pause                          \n\t"
                   "    cmpl $1, (%0)                  \n\t"
                   "    jz 2b                          \n\t"
                   "    jmp 1b                         \n\t"
                   "3:                                 \n\t"
                   : : "r"(&data) : "cc", "%eax", "memory");
   }

   void unlock() {
      asm volatile("xrelease movl $0,(%0) \n\t"
                   : : "r"(&data) : "cc", "memory");
   }
};
```

Restricted Transactional Memory (RTM)

- begin transaction: `unsigned int _xbegin()`
- commit transaction: `void _xend()`
- test if inside a transaction: `unsigned char _xtest()`
- rollback transaction: `void _xabort(const unsigned int errorCode)`
- RTM transactions can be nested up to a limit (but no partial aborts)
- `errorCode` is 8-bit, becomes available to transaction handling code

Hardware Lock Elision Example

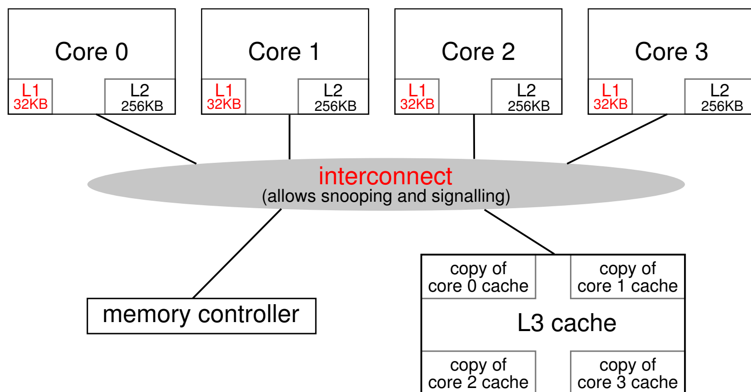
```
struct NaiveRTMTransaction {

   NaiveRTMTransaction() {
      while (true) {
         unsigned status = _xbegin();
         if (status == _XBEGIN_STARTED) {
            return; // transaction started successfully
         } else {
            // on transaction abort, control flow continues HERE
            // status contains abort reason and error code
         }
      }
   }

   ~NaiveRTMTransaction() {
      _xend();
   }
};
```

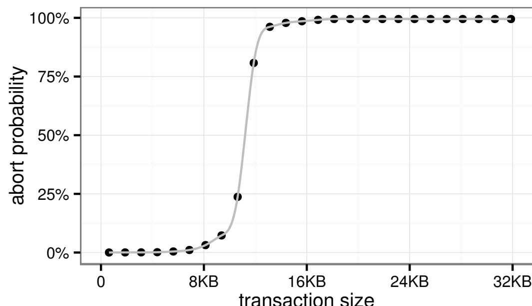
How is HTM implemented?

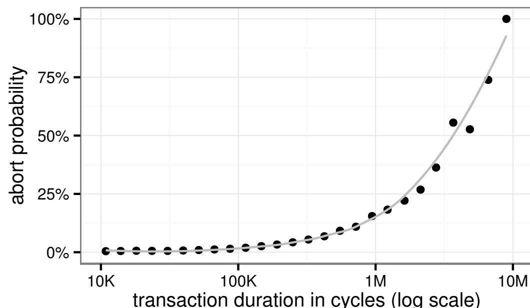
- local L1 cache (32KB) serves as a buffer for transactional writes and for tracking transactional reads at cache line granularity (64 bytes)
- in addition to the L1, there is a larger read set tracking mechanism (similar to a bloom filter)
- cache coherency protocol is slightly modified (e.g., extra T bit for each cache line?) in order to detect read/write and write/write conflicts



Limitations of Haswell's HTM

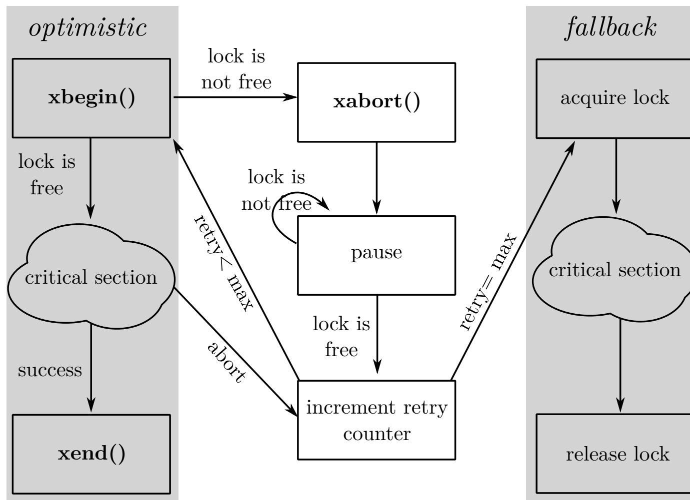
- size (32KB) and associativity (8-way) of L1 cache limit transaction size
- interrupts, context switches limit transaction duration
- certain, rarely used instructions (and `pause`) always cause abort
- due the associativity and the birthday paradox the effective transactions size is smaller than 32KB (in particular for random writes)





Lock Elision with RTM (1)

- due to the hardware limitations one *must* implement a non-transactional fallback path when using RTM (aborts may be deterministic/non-transient)



Lock Elision with RTM (2)

```
std::atomic<int> fallBackLock(0);

struct RTMTransaction {
   RTMTransaction(int max_retries = 5) {
      int nretries = 0;
      while (true) {
         ++nretries;
         unsigned status = _xbegin();
         if (status == _XBEGIN_STARTED) {
            if (fallBackLock.load() == 0) // must add lock to read set
               return; // successfully started transaction
            _xabort(0xff); // abort with code 0xff
         }
         // abort handler
         if ((status & _XABORT_EXPLICIT) && (_XABORT_CODE(status)==0xff)
             && !(status & _XABORT_NESTED)) {
            while (fallBackLock.load()==1) { _mm_pause(); }
         }
         if (nretries>=max_retries) break;
      }
      fallbackPath();
   }
```

Lock Elision with RTM (3)

```
    void fallbackPath() {
        int expected = 0;
        while (!fallBackLock.compare_exchange_strong(expected, 1)) {
            do { _mm_pause(); } while (fallBackLock.load()==1);
            expected = 0;
        }
    }

    ~RTMTransaction() {
        if (fallBackLock.load()==1)
            fallBackLock = 0; // fallback path
        else
            _xend(); // optimistic path
    }
}; // struct RTMTransaction
```

How to Effectively Use HTM?

- once a cache line has been accessed, it cannot be removed from the transactional read/write set
- as a result, lock coupling does not help reducing transaction footprint
- generally, one should use coarse-grained, elided HTM locks
- large data structures may be too large for effective HTM use
- ideal transaction granularity: not too small (transaction overhead), not too large (aborts)

HTM Summary

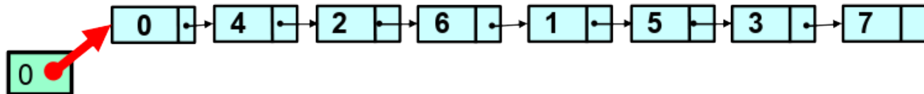
+ easy to use
+ often good scalability (if transactions are not too large)
+ no special memory reclamation necessary
– scalability issues are hard to debug
– HLE is backwards-compatible but not scalable on older CPUs

- more information: Intel® 64 and IA-32 Architectures Optimization Reference Manual, Chapter 14

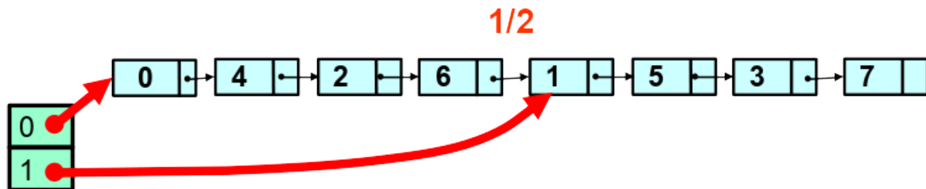
Lock-Free Hash Table: Split-Ordered List

- operations: insert(key, value), remove(key), lookup(key)
- all entries are stored in a single lock-free list (see lock-free list-based set) sorted by the hash value of the key
- dynamically growing array of pointers provides short cuts into this list to get expected $O(1)$ performance
- how to grow the array while other threads are active?

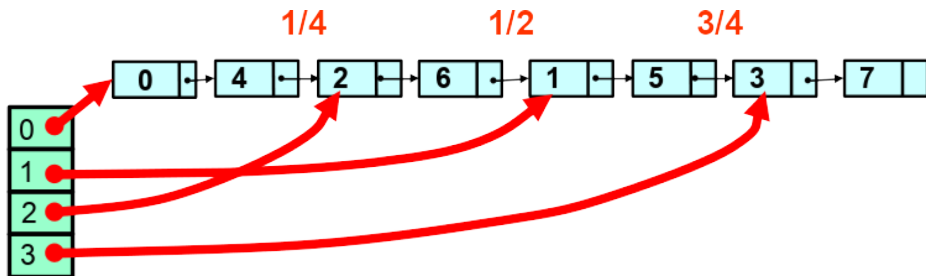
Recursive Split Ordering (1)



Recursive Split Ordering (1)

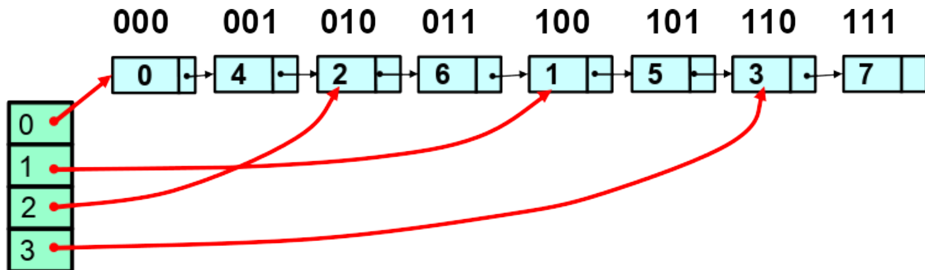


Recursive Split Ordering (1)



Recursive Split Ordering (2)

- key idea: store keys by bit-wise *reverse* hash



Insert

- array grows (by factor of two) when load factor becomes too high
- after growing, the items are not eagerly reassigned
- instead, buckets are lazily initialized on first access (by creating new shortcuts from parent list entry/entries)
- an operation may trigger $log_2(n)$ splits (but $O(1)$ expected)

How to Implement the Array?

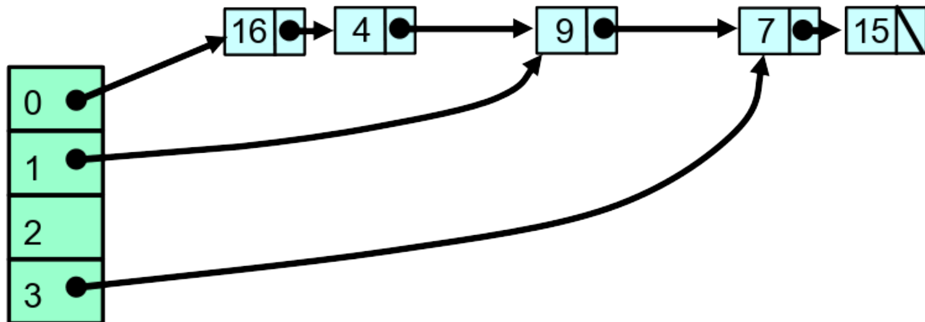
- two-level structure:
  - fixed-size dictionary of (at most) 64 pointers
  - array chunks of size 1, 2, 4, etc.
- enables constant-time random access using lzcnt
- initialize to 0 (e.g., using memset or mmap)

How to Swap Bits?

- there is a hardware instruction for swapping bytes (`uint64_t __builtin_bswap64(uint64_t)`), but not bits
- a fairly efficient way is to read each byte individually and use a (pre-computed) lookup table for swapping each byte

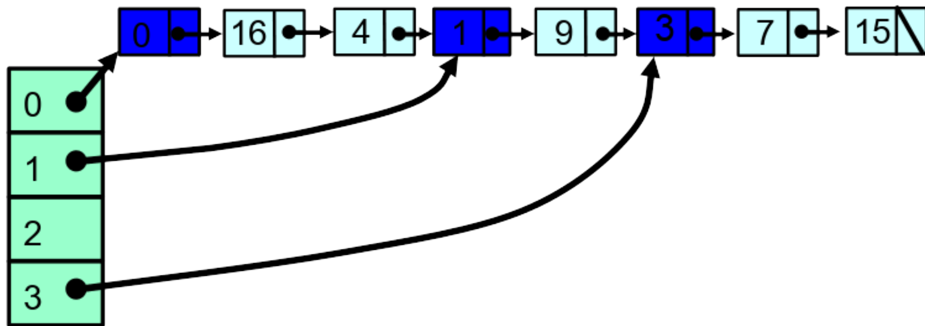
Deletion

- problem: deleting a node using CAS pointed to from a bucket does not work (because it is also being pointed to from the list)



Deletion

- problem: deleting a node using CAS pointed to from a bucket does not work (because it is also being pointed to from the list)
- solution: for every split bucket, insert a special sentinel node into the list



References

- *The art of multiprocessor programming*, Herlihy and Nir Shavit, Morgan Kaufmann, 2008
- *The adaptive radix tree: ARTful indexing for main-memory databases*, Leis and Kemper and Neumann, ICDE 2013
- *The ART of practical synchronization*, Leis and Scheibner and Kemper and Neumann, DaMoN 2016
- *The Bw-Tree: A B-tree for new hardware platforms*, Levandoski and Lomet and Sengupta, ICDE 2013
- *Cache craftiness for fast multicore key-value storage*, Mao and Kohler and Morris, EuroSys 2012
- *Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects*, Michael, TPDS 2004
- *Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems*, Cha and Hwang and Kim and Kwon, VLDB 2001