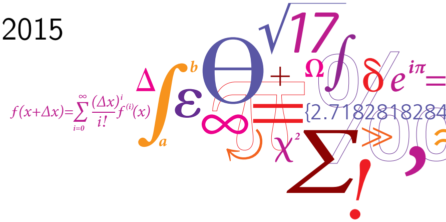
BT-trees

Designing for Hardware Transactional Memory

Lars Bonnichsen
DTU, Denmark

March 26, 2015



Motivation

Writing efficient parallel software is important and difficult:

- Traditional solution: use fine grained synchronization, and minimize synchronization overhead
- Hardware transactional memory (HTM) promises to be simpler, but it shown mixed results performance wise.
    - How can we exploit HTM most efficiently?
    - Can we benefit from HTM? (should we care about it?)

Contributions

You can get extremely good results from HTM, if you reason about how it works

- I suggest 5 guidelines for exploiting HTM
  - Most of which are also good guidelines for traditional synchronization
- With the guidelines I developed an HTM-aware data structure, BT-trees:
  - 3x improvement over state of the art solutions without HTM
  - 2x improvement over state of the art solutions with HTM

Overview

- Introduction
- Transactional memory (Background)
- Designing for HTM
- The design of BT-trees
- Evaluation

Transactional memory

Transactions can implement arbitrary atomic regions

- Example, two threads increment random counters atomically:

```
int counters[32];
#pragma omp parallel 2
{
    i = randomInt(0, 31);
    atomic { counters[i]++; } // Transaction
}
assert(sum(counters, 32) == 2);
```

- Conflict if the threads concurrently increment the same counter.
- Conflicts cause transactions to roll back all writes.

Transactional memory (2)

Transactions can implement arbitrary atomic regions

- Example, two threads increment random counters atomically:

```
char counters* = calloc(32, 1);
#pragma omp parallel 2
{
    i = randomInt(0, 31);
    atomic { counters[i]++; } // Transaction
}
assert(sum(counters, 32) == 2);
```

- Transactions do not require manually allocating locks.
- Transactions can give a speedup in this example, unlike locks.
- Software transactional memory is slow.

Hardware Transactional Memory (HTM)

- Available in Intel Haswell, IBM Power 8.
- Relatively low overhead ($\approx$ 60 cycles)
- Limited. Rolls back on: interrupts, system calls, large transactions, debugger breakpoints, page faults, cache evictions, TLB misses, or problematic instructions, such as division.

Roll backs for reasons other than than conflicts or interrupts are deterministic

- Retrying such transactions is guaranteed to fail
- Normal code assumes that page faults eventually complete

```
atomic { counters[i]++; } // Transaction
```

Synchronization

SLR lock-elision implement locks with a HTM fast path:

- On `lock()`, try to start a transaction
- On repeated transaction roll backs, acquire an actual lock
- On `unlock()` abort transaction if the lock is held.

We use one lock per BT-tree, and hope it is enough to avoid roll backs

The 5 sins of transactions

Well behaved transactions only fail from:

1. False sharing
2. Writing to many cache lines
3. System calls and page faults
4. Reading frequently written cache lines
5. Letting a transaction continue long after its first visible write

The first 3 "sins" are bad under traditional synchronization, but worse (cause aborts) under HTM.

The 4th "sin" may abort other transactions by causing conflicts.

The 5th "sin" may abort this transaction due to conflicts.

The 5 sins of transactions (2)

Well behaved transactions only fail from:

1. False sharing
2. Writing to many cache lines
3. System calls and page faults
4. Reading frequently written cache lines
5. Letting a transaction continue long after its first visible write

Sin 1, 2, and 3 implies that spatial locality is extra important!

Sin 4 and 5 only occur when concurrent transactions truly share data

- Transactions' parallelism is only limited by true sharing (sin 4-5) and restrictions in transactional memory support (sin 1-3)

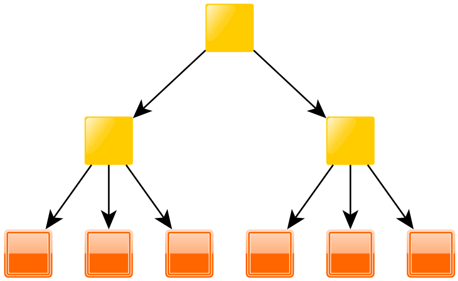
The 5 guidelines for applying transactions

Do what you always do to optimize parallel code, but especially:

1. Optimize spatial locality
2. Avoid reading frequently written data
3. Avoid system calls
4. Minimize time from first visible write to commit
5. Worry less about the size of critical sections

Let us design a concurrent data structure accordingly!

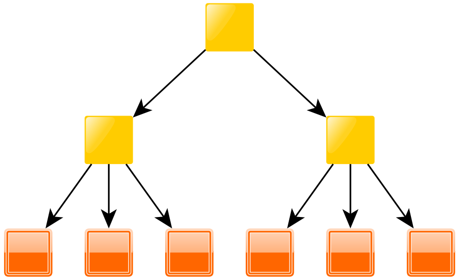
BT-trees



```
class I<K> {
  int size;
  I* c[32];   K k[31];
};
class E<K, V> { K k; V v; };
class L<K, V> { E<K, V> e[32] };
```

- Search tree.
- Internal nodes (`I`) have up to 32 children
- Mapped key-value pairs (`E`) stored in leaf nodes (`L`)
  - Unordered array of 32 key-value pairs.

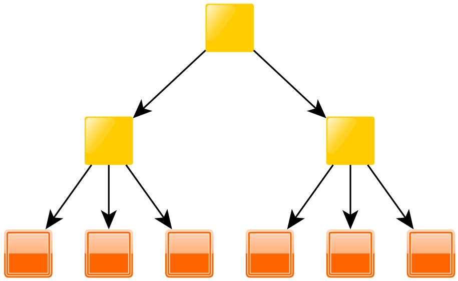
BT-trees balancing



```
class I<K> {
  int size;
  I* c[32];   K k[31];
};
class E<K, V> { K k; V v; };
class L<K, V> { E<K, V> e[32] };
```

- All operations:
  Traverse the tree
  Operate on leaf node
- Balance nodes when we notice they are unbalanced
  - Split full nodes.
  - Merge near-empty nodes.
- Tree grows and shrinks when mergeing and splitting the root

BT-trees features



```
class I<K> {
  int size;
  I* c[32];   K k[31];
};
class E<K, V> { K k; V v; };
class L<K, V> { E<K, V> e[32] };
```

- High spatial locality
- Most operations only write to leaf nodes
- Most reads are in internal nodes
- Balancing is infrequent
- Node operations can be fully unrolled

Experimental setup

Did the redesign improve HTM efficiency?
Compare against:

- STL `map` synchronized with lock-elision
- STL `unordered_map` synchronized with lock-elision

On a machine with recent software:

- GCC 4.9.1, glibc 2.19, Ubuntu LTS Server 14.04.1

And hardware:

- 4-core Intel Xeon E3-1276 v3 @ 3.6 GHz

Experiment

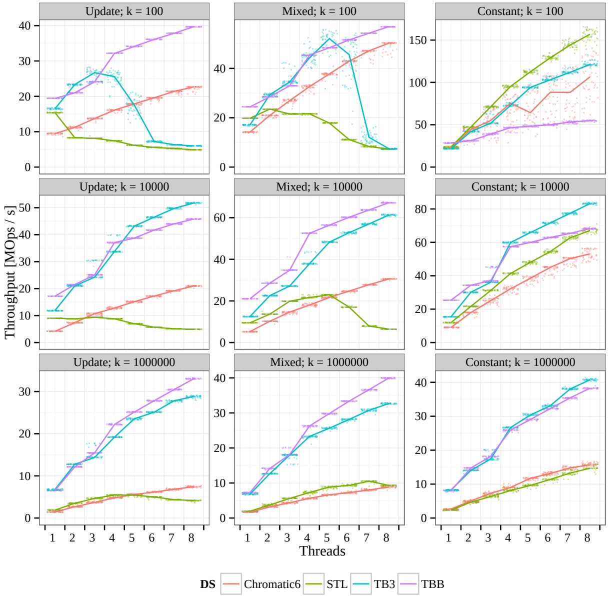
Meassure throughput, energy consumption, contention, *etc* as a function of:

- The number of threads.
- The percentage of *insert*, *remove*, and *search* operations.
- Range of keys (and expected size of the map)

```
prefill(map, (K * pInsert) / (pInsert - pRemove));
#pragma omp parallel
while (hasRunForLessThan5Seconds) {
        key = randomInt(1, K);
        op = randomDouble(0,1);
        if (op < pSearch) map.search(k);
        else if (op > (1 - pInsert)) map.insert(k, v);
        else map.remove(k, v);
}
```

Experiment popularized by PPoPP'2014

Results



- Lemming effect, lock-elision breaks under contention
- BT are generally faster than Chromatic and STL maps, and more scalable than STL maps and TBB

Experimental setup (2)

Is coarse grained application of HTM competetive?
Compare against:

- `Chromatic6`, a state of the art lock-free relaxed red-black tree
- Java `ConcurrentSkipListMap`

On the same machine:

- GCC 4.9.1, glibc 2.19, Ubuntu LTS Server 14.04.1, Oracle JDK 1.8.20, Oracle Server JRE 1.8.20

Experimental setup (3)

Are BT-trees competetive with unordered maps?
Compare against:

- Intel TBB `concurrent_hash_map`
- Java `ConcurrentHashMap`
- Scala `TrieMap` (aka CTries), a state of the art lock-free hash trie

On the same machine:

- GCC 4.9.1, glibc 2.19, Ubuntu LTS Server 14.04.1, Oracle JDK 1.8.20, Oracle Server JRE 1.8.20,

Conclusion

- BT-trees are fast because they exploit speculation.
  - Multiway trees allow for intra thread speculation
- Much more scalable than than other HTM based maps
  - Frequently read memory locations change infrequently