Faster Concurrent Range Queries with Contention Adapting Search Trees Using Immutable Data

Kjell Winblad

Department of Information Technology
Uppsala University, Sweden

2017 Imperial College Computing Student Workshop (ICCSW’2017), Sep 26-27, 2017
Overview

- Background (Concurrent Data Structures)
- Contribution (New concurrent data structure)
- Related Work and Evaluation
- Conclusion
- Questions
Parallel processors (multicores) are everywhere
Parallel processors (multicores) are everywhere

Scalable Concurrent Data Structures:

- Queues, Priority Queues, Sets, etc
- Operating systems, Databases, Parallel algorithms
- Fine-grained locks, lock-free techniques

Our focus:
Ordered Sets with support for linearizable range queries
Background

- Parallel processors (multicores) are everywhere
- Scalable Concurrent Data Structures:
  - Queues, Priority Queues, Sets, etc
Background

- Parallel processors (multicores) are everywhere
- Scalable Concurrent Data Structures:
  - Queues, Priority Queues, Sets, etc
  - Operating systems, Databases, Parallel algorithms
Background

- Parallel processors (multicores) are everywhere
- Scalable Concurrent Data Structures:
  - Queues, Priority Queues, Sets, etc
  - Operating systems, Databases, Parallel algorithms
  - fine-grained locks, lock-free techniques
Parallel processors (multicores) are everywhere

Scalable Concurrent Data Structures:
- Queues, Priority Queues, Sets, etc
- Operating systems, Databases, Parallel algorithms
- fine-grained locks, lock-free techniques

Our focus:
Ordered Sets with support for linearizable range queries
Concurrent Ordered Sets with support for linearizable range queries

- **Ordered set**
  - Represents a set of items with ordered keys
  - Operations: insert item, remove item, lookup item
Concurrent Ordered Sets with support for linearizable range queries

- Ordered set
  - Represents a set of items with ordered keys
  - Operations: insert item, remove item, lookup item

- Linearizable range query operation
  - atomic snapshot of all entries with keys in a range
Concurrent Ordered Sets with support for linearizable range queries

- Ordered set
  - Represents a set of items with ordered keys
  - Operations: insert item, remove item, lookup item
- Linearizable range query operation
  - Atomic snapshot of all entries with keys in a range
- Important for:
  - Big scale databases and data processing platforms
    - Fast updates to store incoming data
    - Concurrent range queries for analytics
Concurrent Ordered Sets with support for linearizable range queries

- Ordered set
  - Represents a set of items with ordered keys
  - Operations: insert item, remove item, lookup item

- Linearizable range query operation
  - atomic snapshot of all entries with keys in a range

- Important for:
  - Big scale databases and data processing platforms
    - Fast updates to store incoming data
    - Concurrent range queries for analytics

- Challenging
  - Large range queries + updates = many collisions
Coarse-grained – Mutable reference to immutable data

- Herlihy [PPoPP’90] (general method)
- SnapTree [PPoPP’10]

Fine-grained – Immutable leaf nodes

- k-ary [PODC’11]
- Leaplist [PODC’13]
Fine-grained vs Coarse-grained – Trade-off

(a) Average #items/query ≈ 5
(b) Average #items/query ≈ 50K

Figure: 10% puts, 10% removes, 55% get, 25% range queries, ≈ 500K items
Contribution

- New concurrent ordered set with linearizable range queries
  - Dynamically adapt the sizes of its immutable parts
    - Contention
    - Number of items covered by range queries
Contribution

- New concurrent ordered set with linearizable range queries
  - Dynamically adapt the sizes of its immutable parts
    - Contention
    - Number of items covered by range queries
  - Attempt to get the best of:
    - Coarse-grained approach
    - Fine-Grained approach
Contribution

- New concurrent ordered set with linearizable range queries
  - Dynamically adapt the sizes of its immutable parts
    - Contention
    - Number of items covered by range queries
  - Attempt to get the best of:
    - Coarse-grained approach
    - Fine-Grained approach
  - Based on:
    Contention Adapting Search Trees (CA trees)
    ISPDC’15, and LCPC’15
CA Tree Structure
CA Tree Animation

[Diagram of CA Tree]

http://www.it.uu.se/research/group/languages/software/ca_tree
CA Tree Animation
CA Tree Animation

CA Tree Animation

http://www.it.uu.se/research/group/languages/software/ca_tree
CA Tree Animation
CA Tree Animation
Range Queries in CA trees *(the old way)*

1. Find base node containing first key in the range
2. Traverse to and lock subsequent base nodes
   - Until base node with key greater than end of range or until the last base node has been found
3. Perform range query in sequential data structures
4. Unlock base node locks and decrease statistics counters if more than one base node

range: [58,67]

Long time period in which conflicts can happen
Range Queries in CA trees (the new way)

- Implement sequential data structure as a mutable reference to an immutable data structure

1. Find base node containing first key in the range
2. Traverse to and lock subsequent base nodes
   - Until base node with key greater than end of range or until the last base node has been found
3. Perform range query in sequential data structures
   Copy references to sequential data structures
4. Unlock base node locks and decrease statistics counters if more than one base node
5. Perform range query in the sequential data structures

😊 Much shorter time period for conflicts
Implementation

- Immutable Treap (balanced binary search tree)
- items stored in fat leaf nodes
  - Up to 64 items in arrays
  - Improves cache locality
How does the new method perform?

(a) Average #items/query ≈ 5  
(b) Average #items/query ≈ 50K

Figure: 10% puts, 10% removes, 55% get, 25% range queries,  
≈ 500K items
Fine-Grained Approach

- **k-ary** – up to k items in immutable leaf nodes
  Brown and Avni [PODC’11]

Coarse-Grained Approach

- **SnapTree** – Fast snapshots based on copy-on-write
  Bronson, Casper, Chafi and Olukotun [PPoPP’10]

Others

- **Old CA tree** – items in mutable Skiplists with fat nodes
  Saganos, and Winblad [LCPC’15]

- **KiWi** – Global version number counter for range queries
  Basin et al. [PPoPP’17]

- See paper for more related work... (Bronson [PPoPP’10],
  Avni et al. [PODC’13], Chatterjee [ICDCN’17])
Evaluation

- Benchmark with a mix of
  - Inserts
  - Removes
  - Lookups
  - Range Queries of size up to max

- Platform
  - NUMA with four Intel Xeon E5-4650 CPUs (2.70GHz)
    8 cores each with hyperthreading = 64 logical cores

- Implementation in Java

- See paper for other benchmark and more data structures...
≈ 500K items, Inserts:10%, Removes:10%, Lookups:55%, Queries:25%-max:10
Medium Sized Range Queries

≈ 500K items, Inserts:10%, Removes:10%, Lookups:55%, Queries:25%-max:1000
Large Range Queries

≈ 500K items, Inserts:10%, Removes:10%, Lookups:55%, Queries:25%–max:100000
Final Remarks

- **CA trees**
  - Key advantage: Adapts to contention and range queries

- **New CA tree variant using Immutable Data**
  - Quick traverse of shared mutable data
    - Conflicts less costly
  - Scales much better than old variants and related work
Contention Statistics Collecting Lock

```c
struct StatLock {
    Lock lock;
    int statistics;
};

void statLock(StatLock slock) {
    if (tryLock(slock.lock)) {
        slock.statistics -= SUCCESS_CONTRIBUTION;
        return;
    }
    lock(slock.lock);
    slock.statistics += FAIL_CONTRIBUTION;
}
```
Contention Adaptation

...  
statLock(base.lock)

performOperation(base.root, parameters...);

if (base.lock.statistics > MAX_CONTENTION) {
    highContentionSplit(tree, base, prevNode);
} else if (base.lock.statistics < MIN_CONTENTION) {
    lowContentionJoin(tree, base, prevNode);
}
statUnlock(base.lock)