More Scalable Ordered Set for ETS Using Adaptation

Kostis Sagonas
Kjell Winblad

Department of Information Technology
Uppsala University, Sweden

5 Sept 2014

What is ETS?

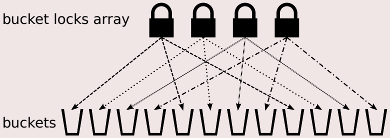
- Erlang Term Storage
- Key-value store
- In-memory database
- Shared memory

ETS Table Types and their Options

- Creates a new shared **hash based** table with **write concurrency** enabled

```
TID1 = ets:new(tab, [set, public,
                     {write_concurrency, true}])
```

- **write concurrency** enables fine grained locking



- Other hash based table types are **bag** and **duplicate bag**

ETS Table Types and their Options

- Creates a new **search tree based** table with **read concurrency** enabled

```
TID2 = ets:new(tab, [ordered_set, public,
                     {read_concurrency, true}])
```

- Current ordered set does not support fine grained locking
- read concurrency enables frequent-read-optimized readers-writer locks
  - One cache line per scheduler
  - Read only operations do not interfere each other

UPPSALA UNIVERSITET

Use of ETS

- Insert an Erlang tuple into a table

```
ets:insert(TID, {42, "A value"}),
```

Use of ETS

- Insert an Erlang tuple into a table

```
ets:insert(TID, {42, "A value"}),
```

- Lookup a value

```
ets:lookup(TID, 42),
```

Use of ETS

- Insert an Erlang tuple into a table

```
ets:insert(TID, {42, "A value"}),
```

- Lookup a value

```
ets:lookup(TID, 42),
```

- Find first element

```
FirstElem = ets:first(TID),
```

- Find next element

```
NextElem = ets:next(TID, FirstElem).
```

Use of ETS

- Insert an Erlang tuple into a table

```
ets:insert(TID, {42, "A value"}),
```

- Lookup a value

```
ets:lookup(TID, 42),
```

- Find first element

```
FirstElem = ets:first(TID),
```

- Find next element

```
NextElem = ets:next(TID, FirstElem).
```

- Many other functions like **delete**, **match**, **foldl**, etc.

Current ETS Scalability Benchmark

Benchmark

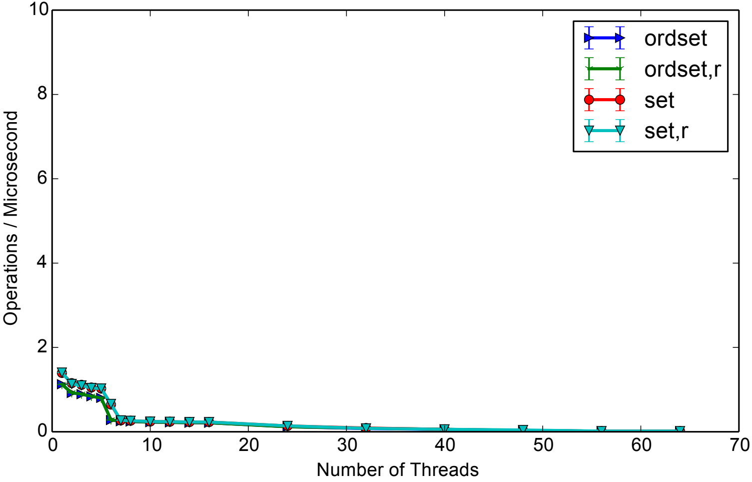
- ets_bench from bencherl
- key range $[1, 2^{21}]$, $2^{21} \approx 2 * 10^6$
- Three phases:
  1. Insert phase: inserts 2^{20} random keys, $2^{20} \approx 10^6$
  2. Update and read phase, parameter for percentage update
  3. Delete phase: deletes 10^6 random keys

Machine

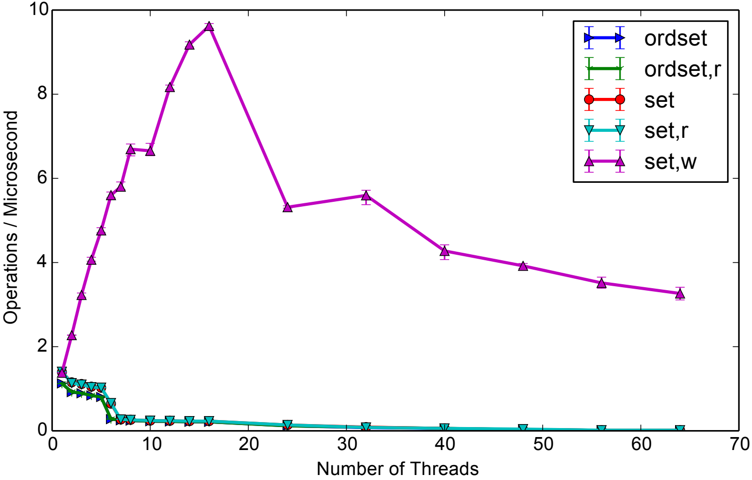
- **Four** Intel(R) Xeon(R) CPU E5-4650 CPUs (2.70GHz), **eight cores each**
  - total of 32 physical cores, each **with hyperthreading**
- The machine has 128GB of RAM and is running Debian Linux 3.10.17-amd64 and **Erlang/OTP release 17.0**

UPPSALA UNIVERSITET

100% Updates

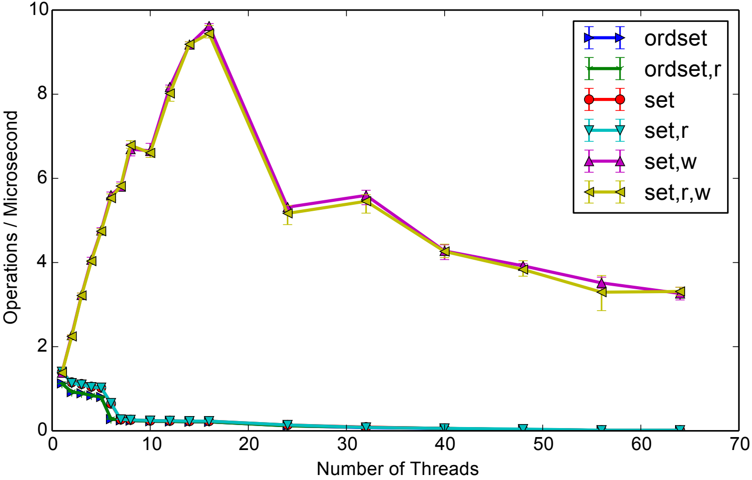


100% Updates

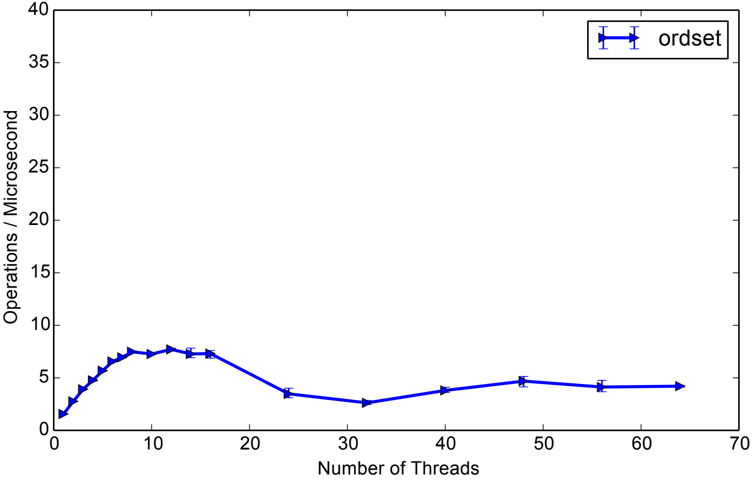

10
8
6
4
2
0
Operations / Microsecond
0
10
20
30
40
50
60
70
Number of Threads
ordset
ordset,r
set
set,r
set,w

UPPSALA UNIVERSITET

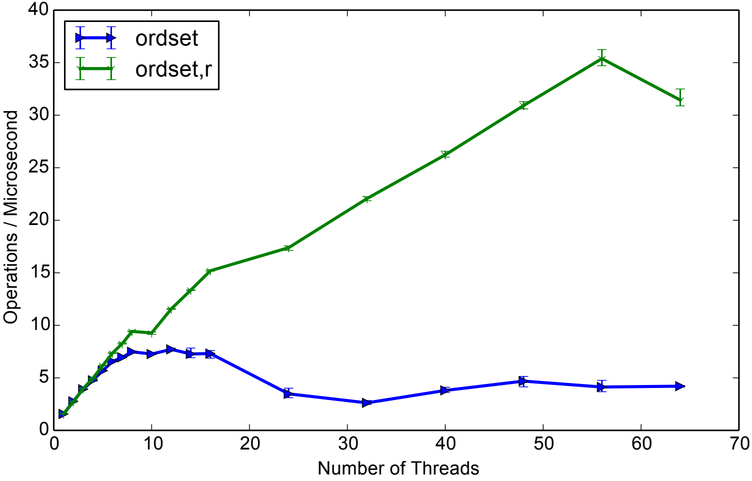
100% Updates



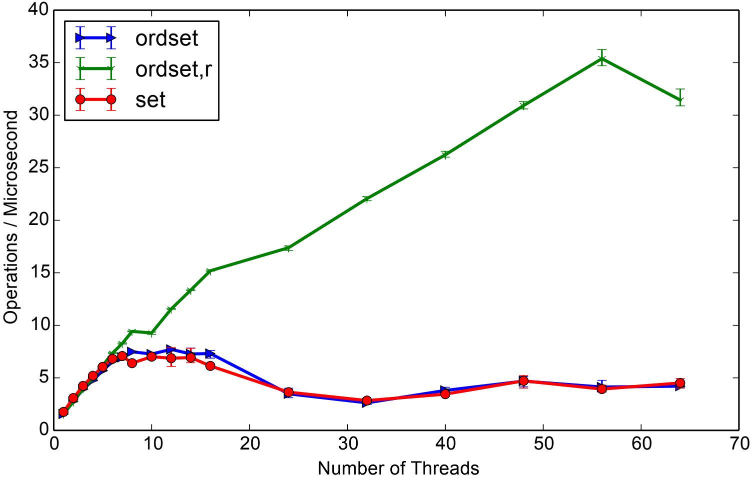
100% Lookups


ordset
Operations / Microsecond
Number of Threads
0
5
10
15
20
25
30
35
40
0
10
20
30
40
50
60
70

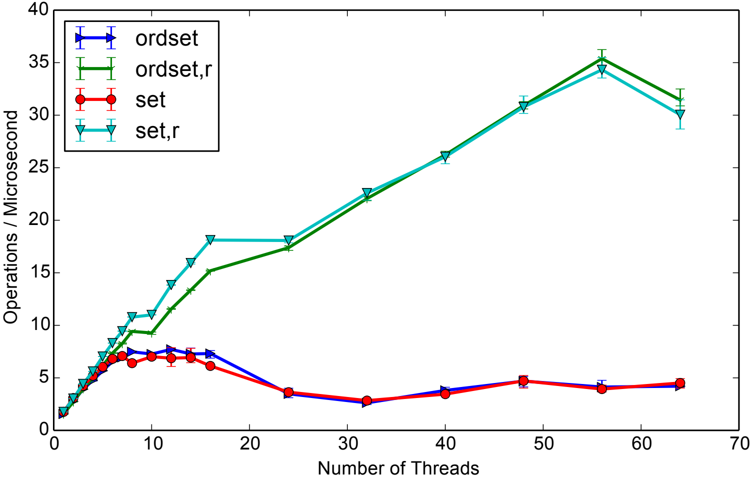
100% Lookups


ordset
ordset,r
Operations / Microsecond
Number of Threads
0
5
10
15
20
25
30
35
40
0
10
20
30
40
50
60
70

100% Lookups

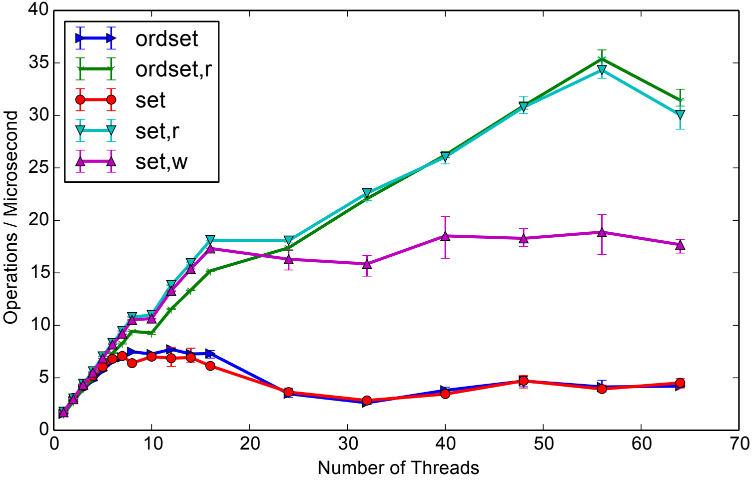

ordset
ordset,r
set
Operations / Microsecond
Number of Threads
0
5
10
15
20
25
30
35
40
0
10
20
30
40
50
60
70

100% Lookups


ordset
ordset,r
set
set,r
Operations / Microsecond
Number of Threads
0
5
10
15
20
25
30
35
40
0
10
20
30
40
50
60
70

UPPSALA UNIVERSITET

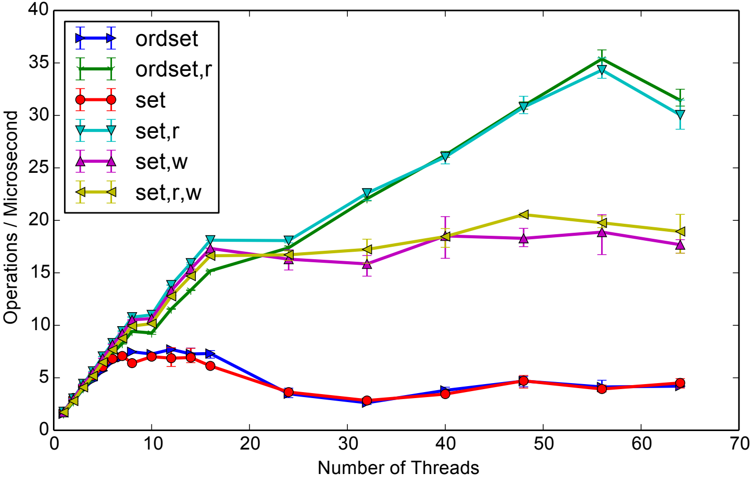
100% Lookups



UPPSALA UNIVERSITET

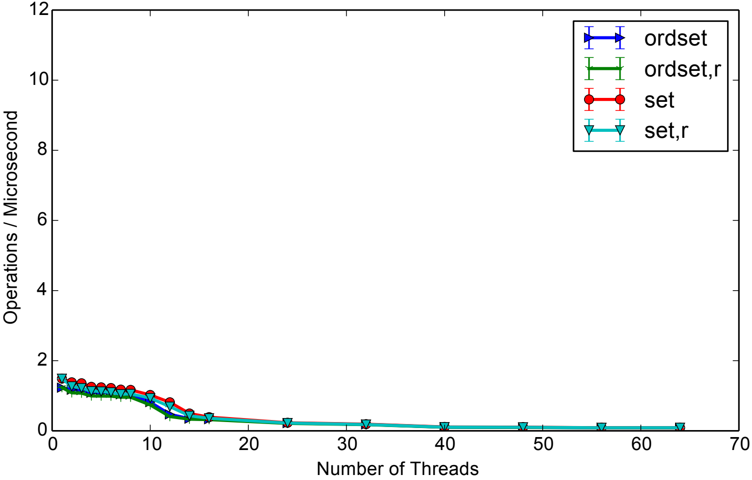
100% Lookups

- Context
- **Motivation**
- CA Trees
- Performance
- Future Work
- Conclusion



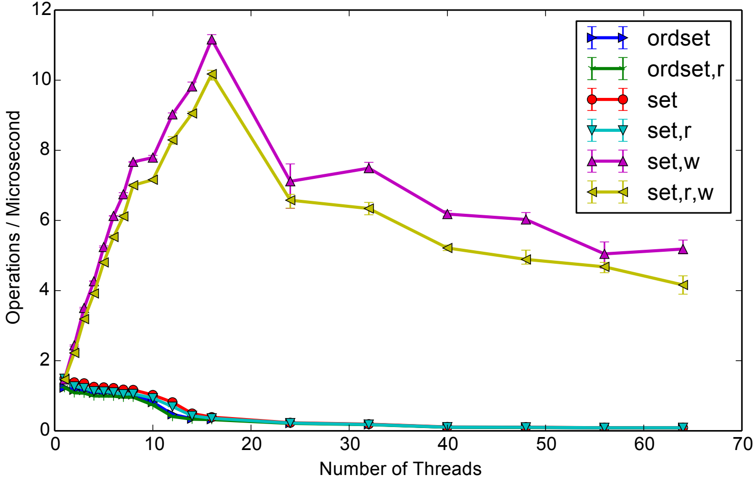
UPPSALA UNIVERSITET

50% Lookups, 50% Updates

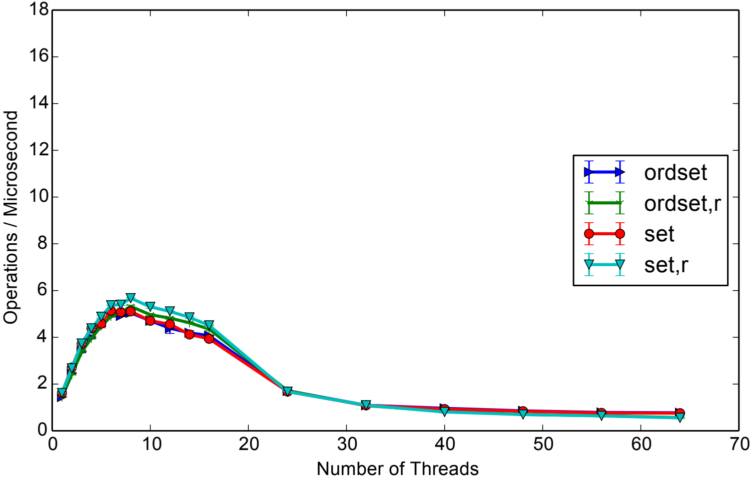


UPPSALA UNIVERSITET

50% Lookups, 50% Updates

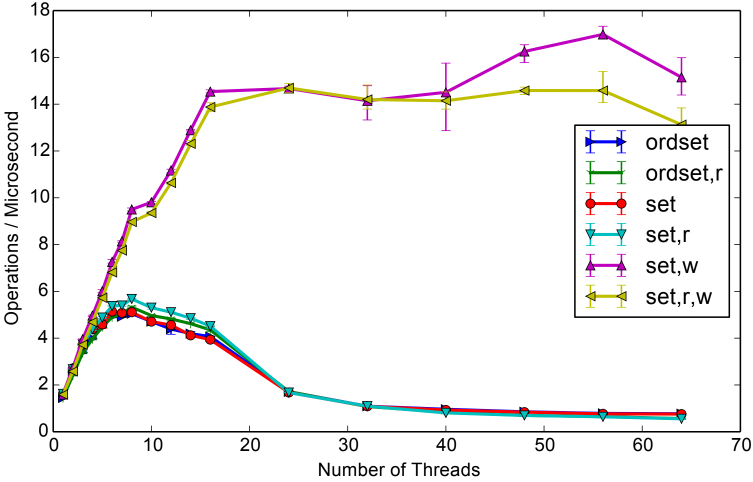


99% Lookups, 1% Updates



UPPSALA UNIVERSITET

99% Lookups, 1% Updates



UPPSALA UNIVERSITET

Summary of Current ETS Scalability

- **Something needs to be done with ordered set to make it scale when there are parallel writes!**
  - Huge slow down even with 99% reads

UPPSALA UNIVERSITET

We Want

Wish list for ordered set

- Good scalability
- Reuse code from the current ETS implementation
- Low overhead in sequential case
    - Current algorithms for concurrent ordered sets sacrifice sequential performance and memory consumption for scalability

Contention Adapting Binary Search Trees (CA trees)

Key Ideas

- Start with sequential binary search tree protected by a lock
- Collect statistics from the lock
- Adapt the tree according to the statistics

UPPSALA UNIVERSITET

CA Tree Components

Statistics Collecting Lock

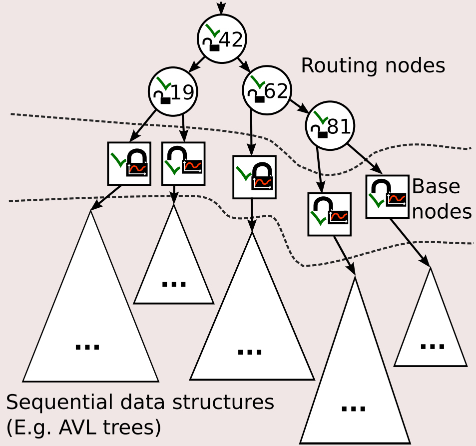
- The lock has an associated counter `C`
- `C += 250` **if** needed to wait to acquire the lock
- `C -= 1` **if not** needed to wait to acquire the lock
- Adapt when `C` reach thresholds
  - E.g. -1000 and 1000

CA Tree Components

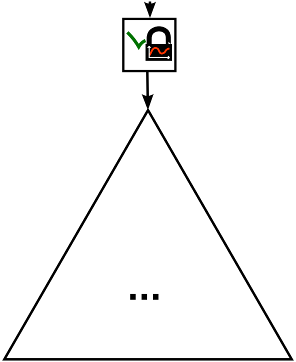
Sequential Ordered Set Data Structure

- Requires support for split and join
  - The **split** operation splits a tree into two so that all keys in one tree are smaller than the keys in the other
  - The **join** operation merges two trees given that all keys in one tree are smaller than the keys in the other
  - $\mathcal{O}(log(N))$ implementations for **AVL trees**, **Red-Black trees**, **Treaps**, etc.

CA Tree Structure

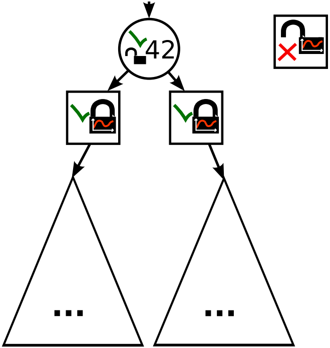


CA Tree Animation

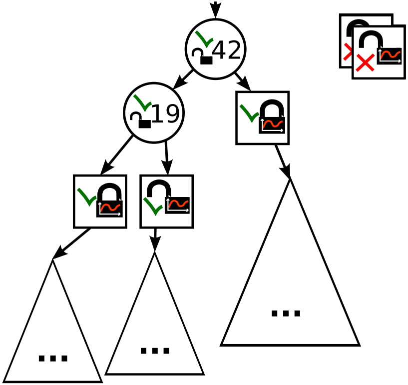


UPPSALA UNIVERSITET

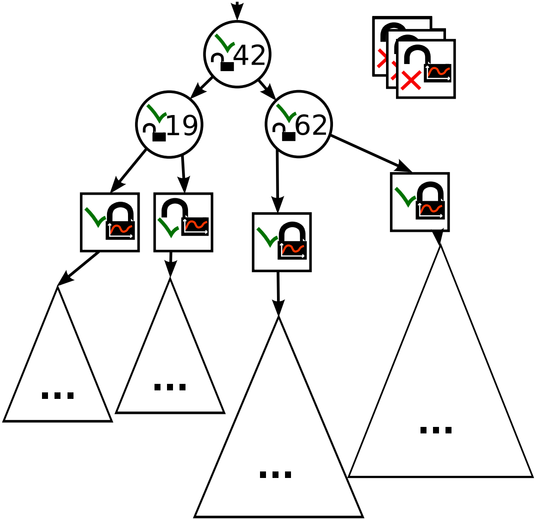
CA Tree Animation



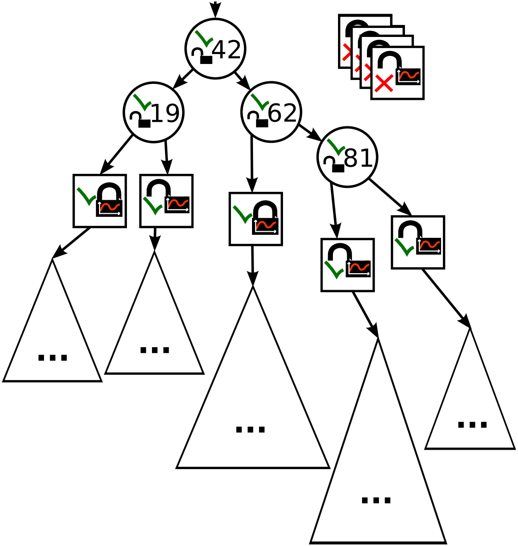
CA Tree Animation



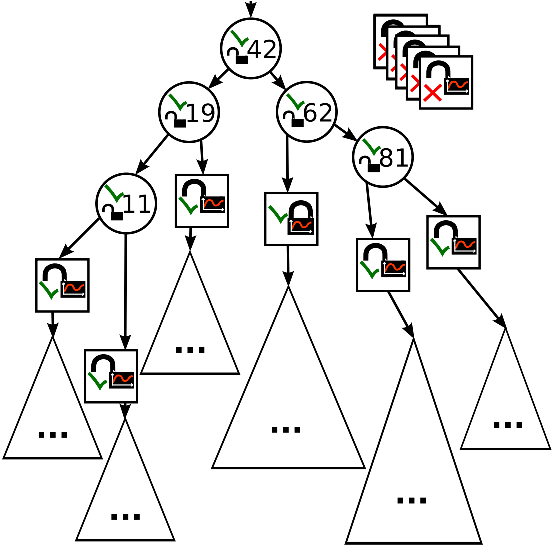
CA Tree Animation



CA Tree Animation

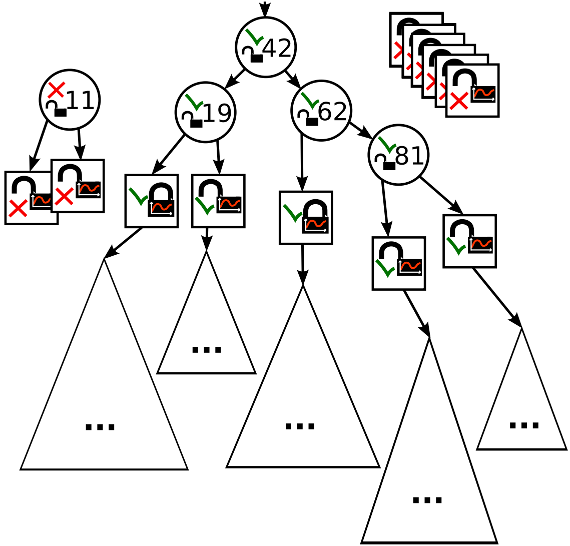


CA Tree Animation

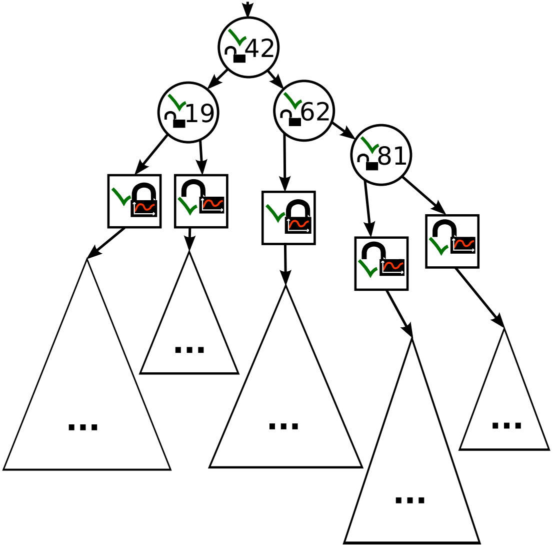


Context
Motivation
CA Trees
Performance
Future Work
Conclusion

CA Tree Animation



CA Tree Animation



Integration into ETS

- Reuse code from current `ordered_set`

Integration into ETS

- Reuse code from current `ordered_set`
- Routing layer needs special memory management
  - Currently quiescent state based reclamation
  - Better to reuse memory reclamation system for lock free data structures that is already integrated into the Erlang runtime system

UPPSALA UNIVERSITET

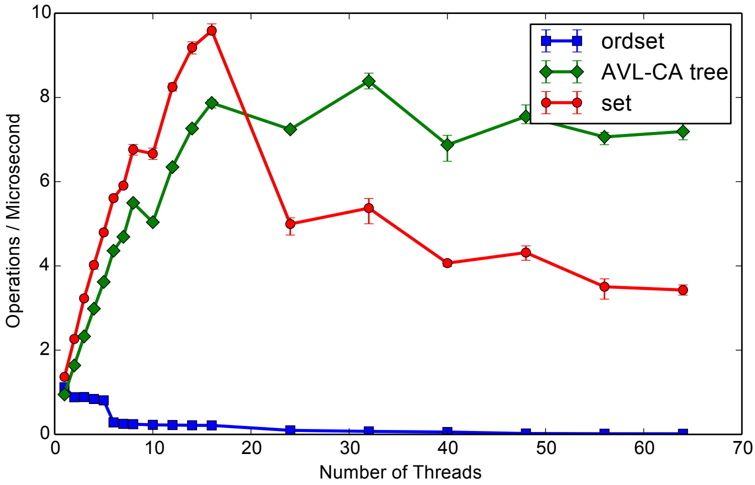
Performance of CA Tree Optimized ETS

The same benchmark again

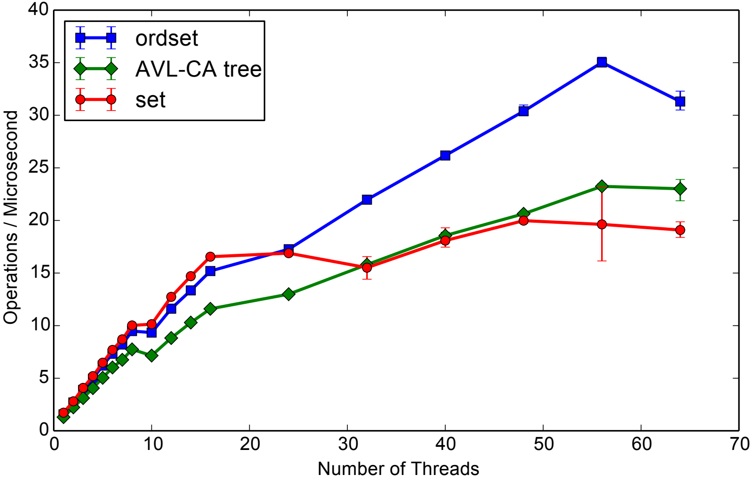
UPPSALA UNIVERSITET

- Context
- Motivation
- CA Trees
- **Performance**
- Future Work
- Conclusion

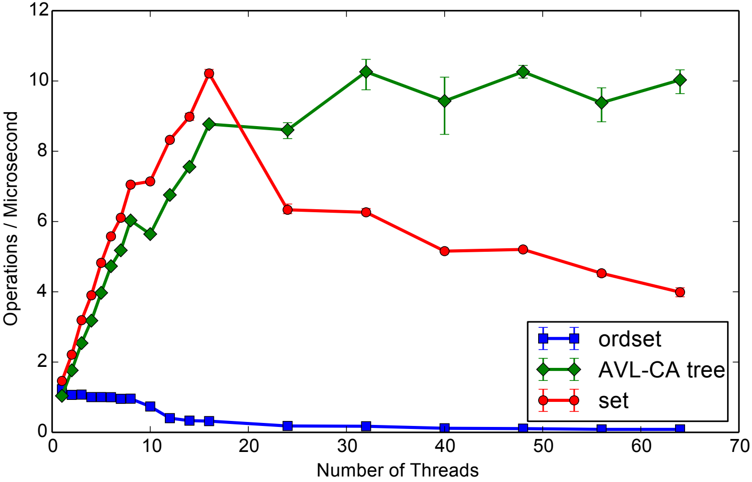
100% Updates



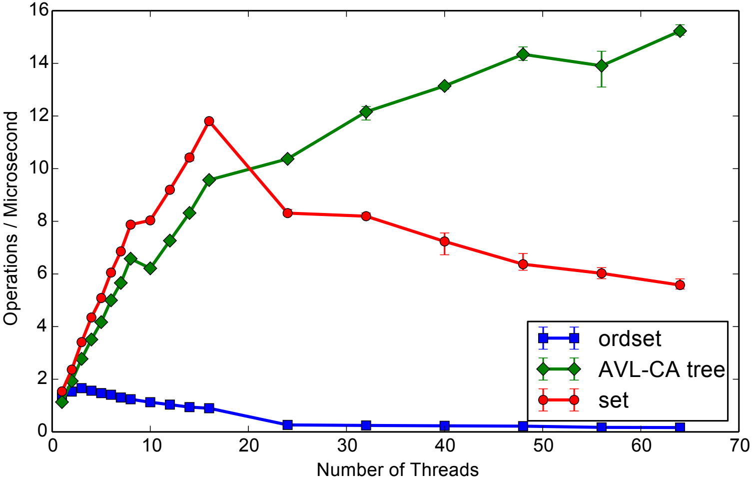
100% Lookups


ordset
AVL-CA tree
set
Operations / Microsecond
Number of Threads
0
5
10
15
20
25
30
35
40
0
10
20
30
40
50
60
70

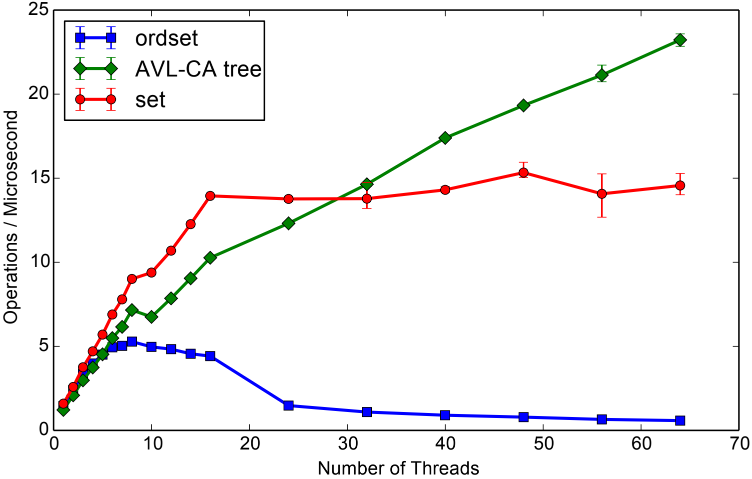
50% Update 50% Lookup Scalabilty



20% Update 80% Lookup Scalabilty



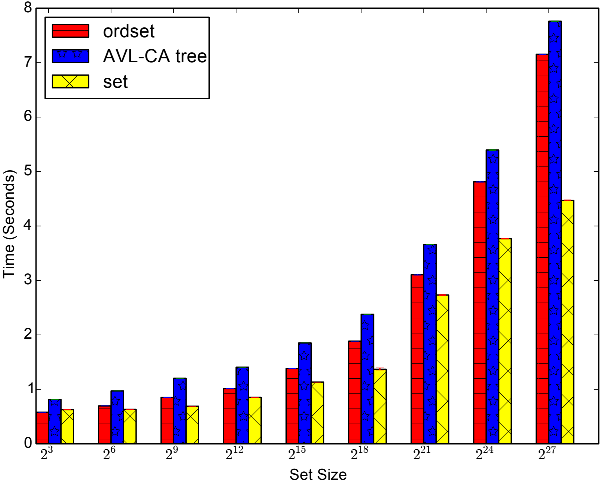
1% Update 99% Lookup Scalabilty



Summary of Scalability Improvements

- The CA trees does not suffer from large slow down
- Scales reasonably well on one chip
- Update heavy scenarios scale far from perfect on NUMA
  - Centralized statistics counter in memory management
  - set has even more problems on NUMA

Sequential Performance, 80% reads



Future Work

To integrate into ETS

- Implement the whole ETS interface
  - Code can be reused from current implementation
- Decide how to integrate it into ETS
  - Always activate on `public` tables
    - Read only case might suffer
  - Only activate when `write_concurrency` is specified

More Information

Work Already Done

- Compare to other concurrent ordered set data structures
- Investigate optimization for read heavy scenarios
  - RW-locks, Sequence Lock, Hardware Lock Elision (HLE)
- Discuss algorithm in detail

More in a Technical Report

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

UPPSALA UNIVERSITET

- Context
- Motivation
- CA Trees
- Performance
- Future Work
- **Conclusion**

Concluding Remarks

- Performance of concurrent writes on ordered set can be substantially improved in ETS
- CA tree based table scales even better than the current hash based implementation on a NUMA system
- good scalability with low sequential overhead

UPPSALA UNIVERSITET

Questions?