Space-Constrained Structures for HTM

Nick Armstrong [1]
narm6003@uni.sydney.edu.au

Vincent Gramoli [1]
vincent.gramoli@sydney.edu.au

Pascal Felber [2]
pascal.felber@unine.ch

[1]The University of Sydney

[2]University of Neuchâtel

25 July, 2016

Introduction

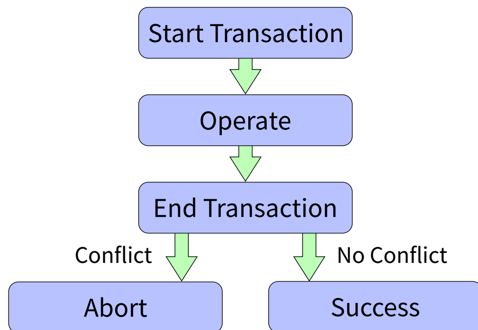
- The Basics
- Data Structures and HTM
- Experiment
- Results and Discussion

Synchronization

- Multiple *threads* run simultaneously
- Ensures resources can be shared without side effects
- Locking and atomic operations
- Transactional memory

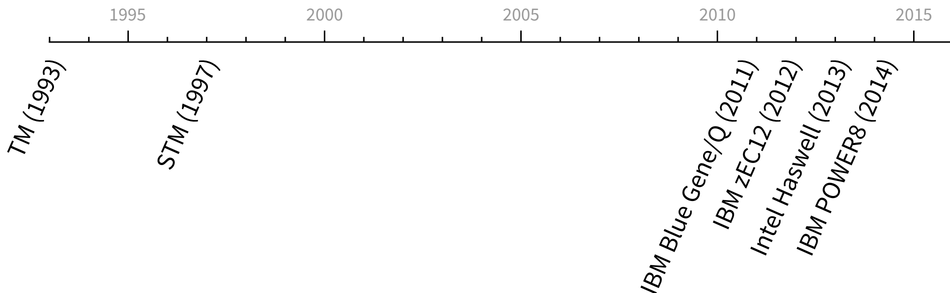
Transactional Memory

- Enables atomic modifications to multiple memory locations
- Keeps track of read sets and write sets
- Threads may execute speculatively
- Conflicts cause transactions to abort
- Implementations exist in both software (STM) and hardware (HTM)



Hardware Transactional Memory

- Transactional memory at the hardware level
- Can be implemented through modifications to cache coherence protocols
- Gives fine grain lock performance with coarse grain semantics
- Much higher performance than STM



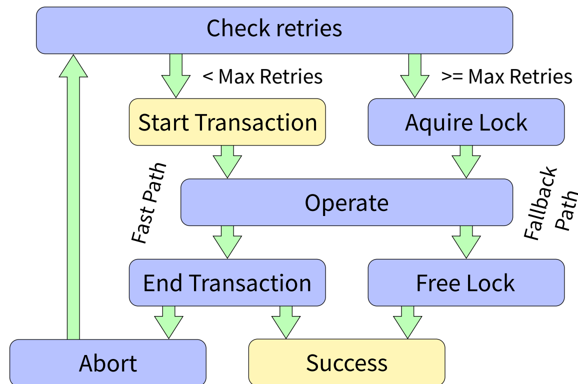
Creating Concurrent Data Structures

- We can use HTM within operations of a data structure
- We can encapsulate entire operations of serial structures
- Just surround an operation with a transaction:

```
tree_htm_op(tree, val) {
    TX_START();
    tree_seq_op(tree, val);
    TX_END;
}
```

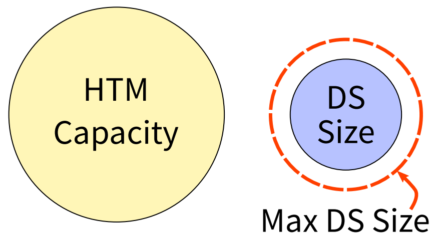
HTM Limitations

- Cache size of CPUs is limited
- Requires a fallback path, possibly locks or STM



Space-Constrained Data Structures

- Structures that can only hold elements in a certain range
- Give an upper bound on number of memory accesses
- Size can be tuned to match HTM capabilities

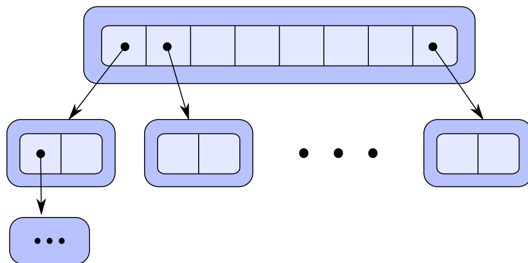


Experiment

- Space-constrained structures may benefit particularly well from HTM
- I implemented a *van Emde Boas tree* (vEB tree) in C
- This implementation is compared against a red black tree using *Synchrobench*
- Utilising HTM on a POWER8 machine

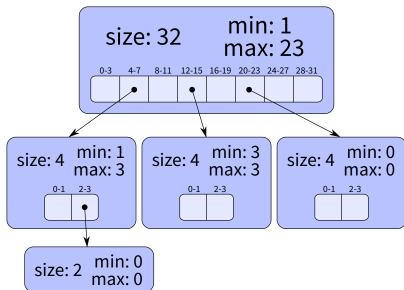
van Emde Boas Trees

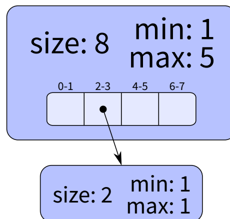
- A recursive ordered tree structure containing elements in range $[0, m-1]$
- $O(m)$ space
- $O(\log \log m)$ insert, delete, contains, next and previous
- $O(1)$ minimum and maximum



van Emde Boas Trees

- A vEB tree of size $m = 2^k$ contains an array of $\sqrt{m}$ vEB trees of size $\sqrt{m}$
- Stores the minimum and maximum values at the root
- Has an auxilliary vEB tree of size $\sqrt{m}$ to keep track of nonempty children.





Synchrobench[1]

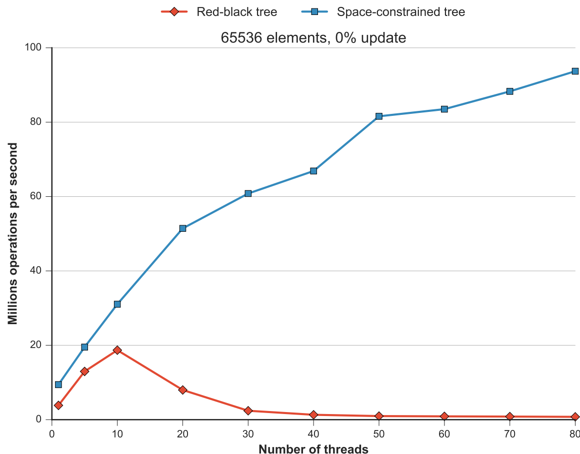
- Micro benchmark suite designed to evaluate concurrent data structures and synchronization techniques
- Used to compare vEB trees to red black trees using HTM
- Parameters used:
  - Size: 65536 (2^{16}), 16777216 (2^{24})
  - Threads: 1, 5, 10, 20, 30, 40, 50, 60, 70 ,80
  - Update rate: 0, 1, 5
  - 4 $\times$ 10000 ms runs per configuration
  - Maximum 15 retries before aquiring fallback lock

[1] https://sites.google.com/site/synchrobench/

Environment

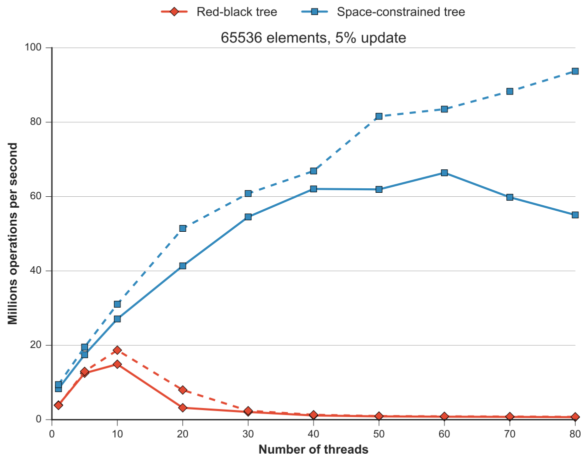
- POWER8 architecture
- Machine with 10 cores, 80 hardware threads
- 32 GB RAM
- Fedora 21

Results



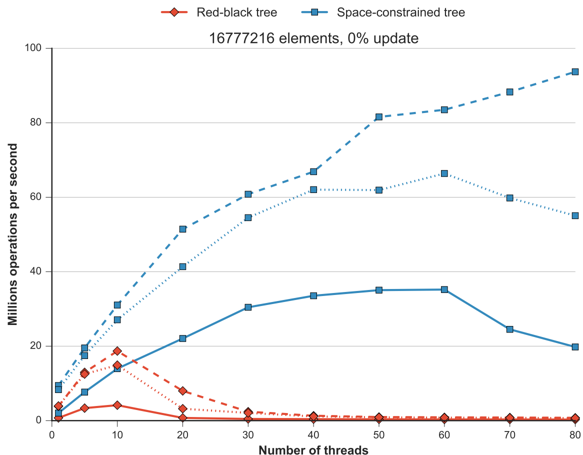
- Red black tree peaks at 10 threads
- vEB tree scales up to 80 threads
- 5.01-fold peak speedup

Results



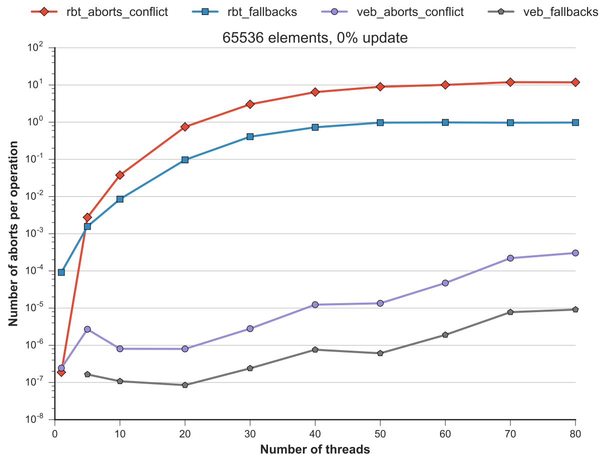
- Throughput decreases with increasing update rate
- Peak performance moves to lower thread counts

Results



- Throughput decreases with increasing size
- vEB trees still scale to many more threads

Results



- 1 in 10 operations acquire the fallback lock for the red black tree at 20 threads
- vEB tree fallback lock aquisitions remain under 1 in 100000 at 80 threads
- With 50+ threads, every red black tree operation falls back to locking

Thank you!

Questions?