No Hotspots Data Structures

Vincent Gramoli (NICTA and U. Sydney)

Joint work with
Tyler Crain (INRIA),
Michel Raynal (U. Rennes, IUF),
Ian Dick (U. Sydney) and
Alan Fekete (U. Sydney)

Dr Vincent Gramoli | Lecturer
School of Information Technologies
Increase in core count

- Pollack’s law [Borkar, CACM 2011]
Problem

Sharing same resource leads to bottlenecks
Same happens between cores sharing data structures (e.g., balanced tree)
Same happens between cores sharing data structures (e.g., balanced tree)
Same happens between cores sharing data structures (e.g., balanced tree)
Same happens between cores sharing data structures (e.g., balanced tree)

Hotspots! Rotations produce contention

Let’s try to remove hotspots!
Removing hotspots

Key ideas:

1. Relaxing invariant (e.g., balanced tree) in case of contention bursts
2. Re-ensuring invariant in the absence of contention
Let’s look at a trendy data structure

- Skip list [Pugh, CACM 1990]
  - Supporting range queries
  - $O(\log n)$ complexity (in expectation)
  - Used by in-memory database companies (memsql, arangodb, etc.)
How does a skip list work?
How does a skip list work?

contains(23)?
How does a skip list work?

Remove(62)
How does a skip list work?

Vincent Gramoli
Concurrent skip lists experience bottlenecks [Fomitchev, Ruppert, PODC’04], [Sundell, Tsigas, SAC’04], [Lea, JSR166]

Hotspots are at the top of the structure towers
Update decoupling

› Eager abstract modification:
  - **Update**: returns after updating the bottom level

› Lazy and selective adaptation:
  - **Update**: postpone the adaptation at higher levels
  - **Remove**: chooses the least likely contended towers
Example: inserting 12

- Eager abstract modification at bottom level

```
insert(12)
```
Example: inserting 12

› Eager abstract modification at bottom level

```
insert(12)
```
Example: inserting 12

- Eager abstract modification at bottom level
- Operation is done, client gets response

```
insert(12)
```
Example: inserting 12

- Eager abstract modification at bottom level
- Operation is done, client gets response
- Lazy update of the higher shortcuts
Example: inserting 12

- Eager abstract modification at bottom level
- Operation is done, client gets response
- Lazy update of the higher shortcuts
Example: removing 36

- Eager abstract modification marks the tower
- Operation is done, client gets response
- Selective adaptation may decide not to remove it
Example: removing 36

- Eager abstract modification marks the tower
- Operation is done, client gets response
- Selective adaptation may decide not to remove it

```
Vincent Gramoli
```

```
remove(36)
```
Non-blocking, no hotspots skip list

Nodes are leaves, IndexItems are internal, access bottom value in $O(1)$

A single maintenance thread restructures continuously in the background

Deletion logically marks node (value=null), insertion unmarks or adds node

Bottom-level list is doubly linked to backtrack to non-marked node
Java Implementation

› Our skip list
- Logarithmic complexity (w/o contention)
- No hotspot: synchronization only at the bottom
SPECjbb 2005

- Base line: `Java.util.concurrent.ConcurrentSkipListMap`

2x 12 core AMD processors SpecJBB [Carlstrom et al. PPoPP’07]
What about porting this skip list in C?

Our skip list was not as efficient on large structure as Fraser’s skip list [K. Fraser, U. Cambridge, PhD thesis, 2003]

Vincent Gramoli
Naïve C port

Cache misses

<table>
<thead>
<tr>
<th>update</th>
<th>0%</th>
<th>10%</th>
<th>30%</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$ size</td>
<td>(7,201K) 0.07%</td>
<td>(7,469K) 0.12%</td>
<td>(7,767K) 0.16%</td>
</tr>
<tr>
<td>$2^{16}$ size</td>
<td>(29,834K) 0.39%</td>
<td>(35,457K) 0.83%</td>
<td>(41,027K) 1.07%</td>
</tr>
<tr>
<td>increase</td>
<td>4.6×</td>
<td>5.9×</td>
<td>5.7×</td>
</tr>
</tbody>
</table>

Although very low, the cache miss rate increases with the structure size.
How to lower-levels rapidly while organizing data contiguously in memory?

Too many indirections require a lot of memory accesses
How to lower-levels rapidly while organizing data contiguously in memory?

Too many indirections require a lot of memory accesses
Array cannot be efficiently updated when the skip list level is decreased
How to lower-levels rapidly while organizing data contiguously in memory?

Too many indirections require a lot of memory accesses
Array cannot be efficiently updated when the skip list level is decreased
Using wheels instead of pointer-based or array-based towers
Actual C implementation

Cache misses (native vs. rotating)

<table>
<thead>
<tr>
<th>update</th>
<th>0%</th>
<th>10%</th>
<th>30%</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>(29,834K) 0.39%</td>
<td>(35,457K) 0.83%</td>
<td>(41,027K) 1.07%</td>
</tr>
<tr>
<td>rotating</td>
<td>(28,061K) 0.29%</td>
<td>(26,258K) 0.37%</td>
<td>(24,094K) 0.38%</td>
</tr>
</tbody>
</table>

This reduces the cache misses rate
Actual C implementation

Performance results

4x 16 core AMD processors

As far as we know this is the most efficient concurrent skip list
Let’s generalize to other data structures

<table>
<thead>
<tr>
<th></th>
<th>Tree</th>
<th>Skip list</th>
<th>Hash table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Invariant</strong></td>
<td>Balance</td>
<td>Level distr.</td>
<td>O(1) load</td>
</tr>
<tr>
<td><strong>Updates</strong></td>
<td>Unbalance</td>
<td>Insert bottom level</td>
<td>Overload</td>
</tr>
<tr>
<td><strong>Restructuring</strong></td>
<td>Rotation</td>
<td>Update higher levels</td>
<td>Resize</td>
</tr>
</tbody>
</table>
Binary search tree synchronized in C with transactional memory

Binary search tree in Java with locks

Skip list in Java with compare-and-swaps

Skip list in C with compare-and-swaps

Most of the code is publicly available: [http://github.com/gramoli/synchrobench](http://github.com/gramoli/synchrobench)

Next step: Hash table?
Conclusion

Contetion-Friendly Non-Blocking Skip List

› Data structures are the **bottlenecks** of modern multi-cores

› **Skip list** is appealing for in-memory database

› **Non-blocking-ness** for fault tolerance and heterogeneity support

› **Contention-friendliness** is the most effective
  - At high level of concurrency
  - Under high contention

› Same trick works for **balanced trees**

› Next steps:
  - Hash table