GETTING TO THE ROOT OF CONCURRENT BINARY SEARCH TREE PERFORMANCE

Maya Arbel-Raviv, Technion
Trevor Brown, IST Austria
Adam Morrison, Tel Aviv U
Optimistic concurrent search trees are crucial in many applications
- (e.g., in-memory databases, internet routers and operating systems)

We want to understand their performance

We study BSTs because there are many variants, making comparisons easier
BST PROTECTED BY A GLOBAL LOCK

Synthetic experiment:
Insert 100,000 keys, then
n threads perform searches

![Graph showing operations per microsecond vs. number of threads]
HAND-OVER-HAND LOCKING (HOH)

Same experiment
LOCKING AND CACHE COHERENCE

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>L2 misses / search</th>
<th>L3 misses / search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global lock</td>
<td>15.9</td>
<td>3.9</td>
</tr>
<tr>
<td>HOH</td>
<td>25.1</td>
<td>7.7</td>
</tr>
</tbody>
</table>

Evict this cache line for all threads!
ACHIEVING HIGH PERFORMANCE

**NO** locking while **searching**!

Example: Unbalanced DGT BST
- Standard BST search
- No synchronization!
## State of the Art BSTs

<table>
<thead>
<tr>
<th>BST</th>
<th>Balanced?</th>
<th>Tree type</th>
<th>Search overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCCO</td>
<td>Y</td>
<td>Internal*</td>
<td>Read <strong>per-node</strong></td>
</tr>
<tr>
<td>DVY</td>
<td></td>
<td>Internal*</td>
<td></td>
</tr>
<tr>
<td>AA</td>
<td></td>
<td>Internal</td>
<td>Write <strong>per-search</strong></td>
</tr>
<tr>
<td>DGT</td>
<td></td>
<td>External</td>
<td></td>
</tr>
<tr>
<td>HJ</td>
<td></td>
<td>Internal</td>
<td>Read <strong>per-node</strong></td>
</tr>
<tr>
<td>RM</td>
<td></td>
<td>Internal</td>
<td>Read <strong>per-search</strong></td>
</tr>
<tr>
<td>NM</td>
<td></td>
<td>External</td>
<td></td>
</tr>
<tr>
<td>EFRB</td>
<td></td>
<td>External</td>
<td></td>
</tr>
</tbody>
</table>

Not covered: lock-free balanced BSTs …
HOW DO THEY PERFORM?
EXPERIMENT: 100% SEARCHES WITH 64 THREADS

![Graph showing search performance across different scenarios with a table below it.]

<table>
<thead>
<tr>
<th></th>
<th>BCCO1</th>
<th>BCCO2</th>
<th>HJ</th>
<th>DVY</th>
<th>RM</th>
<th>AA</th>
<th>NM1</th>
<th>NM2</th>
<th>DGT</th>
<th>EFRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read per-node</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write per-search</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BASIC IMPLEMENTATION ISSUES

Bloated nodes
- Why does node size matter?
- Larger nodes → fewer fit in cache → more cache misses

Scattered fields
- Why does node layout matter?
- Searches may only access a few fields
- Scattered fields → more cache lines → more cache misses

Incorrect usage of C volatile
- Missing volatiles → correctness issue
- Unnecessary volatiles → performance issue
IMPACT OF FIXING THESE ISSUES

Search overhead

<table>
<thead>
<tr>
<th></th>
<th>BCCO1</th>
<th>BCCO2</th>
<th>HJ</th>
<th>DGY</th>
<th>RM</th>
<th>AA</th>
<th>NM1</th>
<th>NM2</th>
<th>DGT</th>
<th>EFRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read per-node</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read per-search</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bloated nodes

|                  | X     | X     |    |     |    |    |     |     |     |      |

Scattered fields

|                  | X     |       |    |     |    |    |     |     |     |      |

Incorrect C volatile

|                  | X     | X     |    |     |    |    |     |     |     |      |

Graph shows M ops/sec for each issue before and after fix.
HOW A FAST ALLOCATOR WORKS

Jemalloc: threads allocate from private **arenas**

Each thread has an arena for each **size class**:
- 8, 16, 32, 48, 64, 80, 96, 128, 192, 256, 320, 384, 448, 512, ...

![Jemalloc per-thread allocation](image)
CACHE LINE CROSSINGS

These nodes cross cache lines!

Fixing bloated nodes can worsen performance!

Not a big deal if the tree fits in the cache

If the tree does not fit in cache, double cache misses for half of your nodes!
Cache is sort of like a hash table

Maps addresses to buckets (4096 for us)

Buckets can only contain up to \( c \) elements (64 for us)

If you load an address, and it maps to a full bucket

- A cache line is evicted from that bucket

“Mod 4096” is not a good hash function

- Patterns in allocation can lead to patterns in bucket occupancy
**Insert** creates a **node** and a **descriptor** (to facilitate **lock-free helping**)

Node size class: 64  Descriptor size class: 64

Which cache sets will these nodes map to?
- Cache indexes used: 0, 2, 4, …      (only **even numbered** indexes)
- Taken modulo 4096, these can only map to **even numbered** cache sets!
- Only half of the cache can be used to store nodes!
SIMPLE FIX: RANDOM ALLOCATIONS

Hypothesis: problem is the rigid even/odd allocation behaviour

Idea: break the pattern with an occasional dummy 64 byte allocation

Fixes the problem!

- Reduces unused cache sets to 1.6%
- Improved search performance by 41%
- ... on our first experimental system, which was an AMD machine.
- However, on an Intel system, this did not improve search performance!
Intel processors prefetch more aggressively

- **Adjacent line prefetcher**: load one extra adjacent cache line
  - Not always the next cache line (can be the previous one)
- **Smallest unit of memory loaded is 128 bytes (two cache lines)**
  - This is also the unit of memory contention
EFFECT OF PREFETCHING ON HJ BST

The occasional dummy allocations break up the even/odd pattern

But...

Whenever search loads a node, it also loads the adjacent cache line
- This is a descriptor or a dummy allocation!
- This is useless for the search
- Only half of the cache is used for nodes

Fix: add padding to nodes or descriptors so they are in different size classes
SEGREGATING MEMORY FOR DIFFERENT OBJECT TYPES

1. Previously described solution
   - Add padding so objects have different size classes

2. A more principled solution
   - Use multiple instances of jemalloc
   - Each instance has its own arenas
   - Allocate different object types from different jemalloc instances

3. An even better solution
   - Use an allocator with support for segregating object types
### PERFORMANCE AFTER ALL FIXES

<table>
<thead>
<tr>
<th></th>
<th>Balanced</th>
<th>Internal</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCCO1</td>
<td>65</td>
<td>60</td>
<td>55</td>
</tr>
<tr>
<td>BCCO2</td>
<td>50</td>
<td>45</td>
<td>40</td>
</tr>
<tr>
<td>HJ</td>
<td>40</td>
<td>35</td>
<td>30</td>
</tr>
<tr>
<td>DVY</td>
<td>35</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td>RM</td>
<td>30</td>
<td>25</td>
<td>20</td>
</tr>
<tr>
<td>AA</td>
<td>25</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>NM1</td>
<td>50</td>
<td>45</td>
<td>40</td>
</tr>
<tr>
<td>NM2</td>
<td>45</td>
<td>40</td>
<td>35</td>
</tr>
<tr>
<td>DGT</td>
<td>35</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td>EFRB</td>
<td>30</td>
<td>25</td>
<td>20</td>
</tr>
</tbody>
</table>

**Search overhead**

- **Read per-node**
- **Read per-search**

**Node size classes**

- Node size class 32 bytes
- Node size class 48 bytes
APPLICATION BENCHMARKS

In-memory database DBx1000 [Yu et al., VLDB 2014]
• Yahoo! Cloud Serving Benchmarks
• TPC-C Database Benchmarks

Used each BST as a database index
• Merges the memory spaces of DBx1000 and the BST!
• Creates similar memory layout issues (e.g., underutilized cache sets)
• And new ones (e.g., scattering of nodes across many pages)
• Even accidentally fixes memory layout issues in DBx1000

See paper for details
RECOMMENDATIONS

When designing and testing a data structure
- Understand your memory layout!
- How are nodes laid out in cache lines? Pages?
- What types of objects are near one another?

When adding a data structure to a program
- You are merging two memory spaces
- Understand your new memory layout!

New tools needed?

http://tbrown.pro
Tutorials, code, benchmarks

Companion talk:
Good data structure experiments are R.A.R.E