Scalable Concurrent Hash Tables
via Relativistic Programming

Josh Triplett

September 24, 2009
Speed of data < Speed of light

- Speed of light: $3 \times 10^8$ meters/second
- Processor speed: 3 GHz, $3 \times 10^9$ cycles/second
- 0.1 meters/cycle (4 inches/cycle)
- Ignores propagation delay, ramp time, speed of signals
Speed of data < Speed of light

- Speed of light: $3 \times 10^8$ meters/second
- Processor speed: 3 GHz, $3 \times 10^9$ cycles/second
- 0.1 meters/cycle (4 inches/cycle)
- Ignores propagation delay, ramp time, speed of signals
- One of the reasons CPUs stopped getting faster
- Physical limit on memory, CPU–CPU communication
Throughput vs Latency

- CPUs can do a lot of independent work in 1 cycle
- CPUs can work out of their own cache in 1 cycle
- CPUs can’t communicate and agree in 1 cycle
How to scale?

• To improve scalability, work independently
• Agreement represents the bottleneck
• Scale by reducing the need to agree
Classic concurrent programming

- Every CPU agrees on the order of instructions
- No tolerance for conflicts
- Implicit communication and agreement required
- Does not scale
- Example: mutual exclusion
Relativistic programming

- By analogy with physics: no global reference frame
- Allow each thread to work with its observed “relative” view of memory
- Minimal constraints on instruction ordering
- Tolerance for conflicts: allow concurrent threads to access shared data at the same time, even when doing modifications.
Why relativistic programming?

- Wait-free
- Very low overhead
- Linear scalability
Concrete examples

- Per-CPU variables
Concrete examples

- Per-CPU variables
- Deferred destruction — Read-Copy Update (RCU)
What does RCU provide?

- Delimited readers with near-zero overhead
- “Wait for all current readers to finish” operation
- Primitives for conflict-tolerant operations:
  - rcu_assign_pointer
  - rcu_deference
What does RCU provide?

- Delimited readers with near-zero overhead
- “Wait for all current readers to finish” operation
- Primitives for conflict-tolerant operations: `rcu_assign_pointer`, `rcu_deference`
- Working data structures you don't have to think hard about
RCU data structures

- Linked lists
- Radix trees
- Hash tables, sort of
Hash tables, sort of

- RCU linked lists for buckets
- Insertion and removal
- No other operations
New RCU hash table operations

- Move element
- Resize table
Move operation

```
<table>
<thead>
<tr>
<th>a</th>
<th>→</th>
<th>n₁</th>
<th>→</th>
<th>n₂</th>
<th>→</th>
<th>n₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>→</td>
<td>n₄</td>
<td>→</td>
<td>n₅</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Key: "old"
Move operation

```
<table>
<thead>
<tr>
<th></th>
<th>n1</th>
<th>n2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

n4  n5  n3

key: "new"
```
Move operation semantics

- If a reader doesn’t see the old item, subsequent lookups of the new item must succeed.
- If a reader sees the new item, subsequent lookups of the old item must fail.
- The move operation must not cause concurrent lookups for other items to fail.
- Semantics based roughly on filesystems
Move operation challenge

- Trivial to implement with mutual exclusion
  - Insert then remove, or remove then insert
  - Intermediate states don’t matter
- Hash table buckets use linked lists
- RCU linked list implementations provide insert and remove
- Move semantics not possible using just insert and remove
Current approach in Linux

- Sequence lock
- Readers retry if they race with a rename
- Any rename
Solution characteristics

• Principles:
  • One semantically significant change at a time
  • Intermediate states must not violate semantics
• Need a new move operation specific to relativistic hash tables, making moves a single semantically significant change with no broken intermediate state
• Must appear to simultaneously move item to new bucket and change key
Solution characteristics

• Principles:
  • One semantically significant change at a time
  • Intermediate states must not violate semantics

• Need a new move operation specific to relativistic hash tables, making moves a single semantically significant change with no broken intermediate state

• Must appear to simultaneously move item to new bucket and change key
• Cross-link end of new bucket to node in old bucket
• Cross-link end of new bucket to node in old bucket
• While target node appears in both buckets, change the key
Key idea

- Cross-link end of new bucket to node in old bucket
- While target node appears in both buckets, change the key
- Need to resolve cross-linking safely, even for readers looking at the target node
- First copy target node to the end of its bucket, so readers can’t miss later nodes
- Memory barriers
Benchmarking with rcuhashbash

• Run one thread per CPU.
• Continuous loop: randomly lookup or move
• Configurable algorithm and lookup:move ratio
• Run for 30 seconds, count reads and writes
• Average of 10 runs
• Tested on 64 CPUs
Results, 999:1 lookup:move ratio, reads

Millions of Hash Lookups per Second

CPUs

Proposed algorithm
Current Linux (RCU+seqlock)
Per-bucket spinlocks
Per-bucket reader-writer locks
Results, 1:1 lookup:move ratio, reads

- Per-bucket spinlocks
- Per-bucket reader-writer locks
- Proposed algorithm
- Current Linux (RCU+seqlock)

The graph shows the performance of different locking mechanisms in terms of millions of hash lookups per second, measured across a range of CPUs from 1 to 64.
Resizing RCU-protected hash tables

- Disclaimer: work in progress
- Working on implementation and test framework in rcuhashbash
- No benchmark numbers yet
- Expect code and announcement soon
Resizing algorithm

- Keep a secondary table pointer, usually NULL
- Lookups use secondary table if primary table lookup fails
• Keep a secondary table pointer, usually NULL
• Lookups use secondary table if primary table lookup fails
• Cross-link tails of chains to second table in appropriate bucket
Resizing algorithm

- Keep a secondary table pointer, usually NULL
- Lookups use secondary table if primary table lookup fails
- Cross-link tails of chains to second table in appropriate bucket
- Wait for current readers to finish before removing cross-links from primary table
Resizing algorithm

- Keep a secondary table pointer, usually NULL
- Lookups use secondary table if primary table lookup fails
- Cross-link tails of chains to second table in appropriate bucket
- Wait for current readers to finish before removing cross-links from primary table
- Repeat until primary table empty
- Make the secondary table primary
- Free the old primary table after a grace period
For more information

- Code: git://git.kernel.org/pub/scm/linux/kernel/git/josh/rcuhashbash (Resize coming soon!)
- Relativistic programming: http://wiki.cs.pdx.edu/rp/
- Email: josh@joshtriplett.org