Scalable Data Structures with Relativistic Programming

Josh Triplett

June 3, 2009
Motivations for relativistic programming:
  • Correctness requires performance
  • Performance requires scalability
  • Scale by reducing the need to agree

What does “need to agree” mean?

Why do we think need to agree matters?
• How do we create scalable data structures using relativistic programming techniques?
• How do we test the scalability of these data structures?
• How can we make this process easier in the future?
Hash table example

- Sample to illustrate relativistic programming methodology
- Maps key to value
- Chaining: hash function maps key to bucket with list of nodes containing keys and values
- Lookup: use hash to find bucket, then check key of each node
- Allows lookups in constant average time
- Used frequently in operating systems
Move operation

```
 Move operation
```

```
 a

  ...

 b

\[ n_1 \rightarrow n_2 \rightarrow n_3 \]

\[ n_4 \rightarrow n_5 \]

key

"old"
```
Move operation

\[ \text{Move operation} \]

\[ \text{Diagram showing the move operation} \]

\[ \text{Diagram with nodes labeled a, n_1, n_2, ..., n_3, and a key labeled "new"} \]
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 on filesystems
Why the move operation?

- Trivial to implement with mutual exclusion
  - Insert then remove, or remove then insert
  - Intermediate states don’t matter
- Hash table buckets use linked lists
- Relativistic linked list implementations already exist
- Move operation semantics not implementable using relativistic linked list operations (insert and remove)
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
My solution

- Cross-link end of new bucket to node in old bucket
My solution

- Cross-link end of new bucket to node in old bucket
- While target node appears in both buckets, change the key
• 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 move 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 read or write
- Configurable algorithm and read:write ratio
- Run for 30 seconds, count reads and writes
Results, 999999:1 read:write ratio, reads

The diagram shows the reads for different contexts as a function of the number of CPUs. The contexts include:
- rcu
- rcu_seq
- rwlock
- spinlock
- table_rwlock
- table_spinlock

The x-axis represents the number of CPUs ranging from 1 to 64, and the y-axis represents the number of reads ranging from 0 to 7e+09.
Results, 999999:1 read:write ratio, writes
Results, 999:1 read:write ratio, reads

- rcu
- rcu_seq
- rwlock
- spinlock
- table_rwlock
- table_spinlock

Reads vs CPUs graph showing the performance of different synchronization mechanisms under high read load.
Results, 999:1 read:write ratio, writes

![Graph showing writes vs CPUs for different lock types: rcu, rcu_seq, rwlock, spinlock, table_rwlock, table_spinlock. The y-axis represents writes, and the x-axis represents CPUs. The graph shows a linear increase with an increasing number of CPUs.]
Results, 1:1 read:write ratio, reads

![Graph showing the performance of different synchronization mechanisms (rcu, rcu_seq, rwlock, spinlock, table_rwlock, table_spinlock) under a 1:1 read:write ratio. The graph plots the number of reads against the number of CPUs.]
Results, 1:1 read:write ratio, writes

![Graph showing writes vs CPUs for different locking mechanisms: rcu, rcu_seq, rwlock, spinlock, table_rwlock, table_spinlock.]
Relativistic data structures

- Determine the required semantics and design to those only.
- Within these semantics, reduce algorithmic need to agree.
- One semantically significant change at a time
- Intermediate states must not violate semantics
- Changes potentially become visible to other processors immediately, or after arbitrarily long delay
- Ordering not guaranteed unless explicitly requested
Future data structures

- Resizable hash tables.
- Heaps, priority heaps, balanced trees, skip lists, Judy arrays...
Problems

- Hash table required inventing a new algorithm.
- Coding the algorithm required direct use of low-level operations such as memory barriers.
- How can we make this easier?
Goals

- High-level building blocks for relativistic data structures.
- No inventiveness required to create algorithms for new data structures or semantics.
- No manual placement of memory barriers or other low-level synchronization operations.