Design and Evaluation of Scalable Software

Damian Dechev \(^{1,2}\)

1: University of Central Florida, CSE S3 Lab, http://cse.eecs.ucf.edu
Orlando, FL

2: Sandia Labs,
Livermore, CA
Traditional Scaling Process (La-Z-Boy Approach)

- **User code**
  - 1.8x
  - 3.6x
  - 7x

- **Traditional Uniprocessor**
  - Time: Moore’s law
Multicore Scaling Process

Unfortunately, not so simple...
Real-World Scaling Process

- Speedup: 1.8x, 2x, 2.9x

- User code

- Multicore

Parallelization and Synchronization require great care...
Problem: Concurrent Hash Table

• Problem is
  - `add()`, `remove()`, `contains()`
  - In a set: take time linear in set size

• We want
  - Constant-time complexity (at least, on average)
  - Hash function (`h: items → integers`); elements uniformly distributed
Sequential Hash Map

\[ h(k) = k \mod 4 \]

2 Items

*buckets*

\[ h(k) = k \mod 4 \]
Add an Item

\[ h(k) = k \mod 4 \]

3 Items
Add Another: Collision

h(k) = k mod 4

4 Items
More Collisions

h(k) = k mod 4

5 Items
More Collisions

Problem: buckets getting too long

5 Items

\[ h(k) = k \mod 4 \]
Resizing

Grow the array

\[ h(k) = k \mod 8 \]

5 Items
Resizing

Adjust hash function

\[ h(k) = k \mod 8 \]
Resizing

\[ h(4) = 0 \mod 8 \]

\[ h(k) = k \mod 8 \]
Resizing

\[ h(4) = 4 \mod 8 \]

\[ h(k) = k \mod 8 \]
Resizing

$h(15) = 7 \mod 8$

$h(k) = k \mod 8$
Resizing

\[ h(k) = k \mod 8 \]

\[ h(15) = 7 \mod 8 \]
class SimpleHashSet {
    LockFreeList[] table;

    SimpleHashSet(int capacity) {
        table = new LockFreeList[capacity];
        for (int i = 0; i < capacity; i++)
            table[i] = new LockFreeList();
    }

    ...
Fields

```java
class SimpleHashSet {
    LockFreeList[] table;

    SimpleHashSet(int capacity) {
        table = new LockFreeList[capacity];
        for (int i = 0; i < capacity; i++)
            table[i] = new LockFreeList();
    }

    ...
}
```

Array of lock-free lists
Constructor

```java
class SimpleHashSet {
    LockFreeList[] table;

    SimpleHashSet(int capacity) {
        table = new LockFreeList[capacity];
        for (int i = 0; i < capacity; i++)
            table[i] = new LockFreeList();
    }

    ...
```

Initial size
Constructor

class SimpleHashSet {
    LockFreeList[] table;

    SimpleHashSet(int capacity) {
        table = new LockFreeList[capacity];
        for (int i = 0; i < capacity; i++)
            table[i] = new LockFreeList();
    }

    ...

    Allocate memory
Constructor

class SimpleHashSet {
    LockFreeList[] table;

    SimpleHashSet(int capacity) {
        table = new LockFreeList[capacity];
        for (int i = 0; i < capacity; i++)
            table[i] = new LockFreeList();
    }

    ...

    Initialization
Add Function

```java
boolean add(Object key) {
    int hash =
        key.hashCode() % table.length;
    return table[hash].add(key);
}
```
boolean add(Object key) {
  int hash = 
    key.hashCode() % table.length;
  return table[hash].add(key);
}

Use object hash code to pick a bucket
boolean add(Object key) {
    int hash =
        key.hashCode() % table.length;
    return table[hash].add(key);
}

Call bucket's add() function
No Brainer?

• We just saw a
  - Simple
  - Lock-free
  - Concurrent hash-based set implementation

• What’s not to like?
No Brainer?

• We just saw a
  - Simple
  - Lock-free
  - Concurrent hash-based set implementation
• What’s not to like?
• We don’t know how to resize ...
Set Method Mix

• Typical load
  - 90% contains()
  - 9% add ()
  - 1% remove()

• Growing is important

• Shrinking not so much
Coarse-Grained Locking

• Good parts
  - Simple
  - Hard to mess up
• Bad parts
  - Sequential bottleneck
Fine-Grained Locking

Each lock associated with one bucket
Make sure table reference didn't change between resize decision and lock acquisition.

Acquire locks in ascending order.
Allocate new super-sized table
Striped Locks: each lock now associated with two buckets
Insight

• The contains() method
  - Does not modify any fields
  - Why should concurrent contains() calls conflict?
FIFO R/W Lock

• As soon as a writer requests a lock
• No more readers accepted
• Current readers “drain” from lock
• Writer gets in
Optimistic Synchronization

• If the `contains()` function
  - Scans without locking

• If it finds the key
  - OK to return true

• What if it doesn't find the key?
Optimistic Synchronization

• If it doesn’t find the key
  - May be victim of resizing
• Must try again
  - Getting a read lock this time
• Makes sense if
  - Keys are present
  - Resizes are rare
Stop The World Resizing

- The resizing we have seen up till now stops all concurrent operations
- Can we design a resize operation that will be incremental
- Need to avoid locking the table
- A lock-free table with incremental resizing
Lock-Free Resizing Problem

Need to extend table
Lock-Free Resizing Problem

![Diagram of Lock-Free Resizing Problem]

Need to extend table
Lock-Free Resizing Problem
Lock-Free Resizing Problem

We need a new idea…

to remove and then add even a single item single location CAS not enough
Don’t move the items

- Move the buckets instead
- Keep all items in a single lock-free list
- Buckets become “shortcut pointers” into the list
Recursive Split Ordering
Recursive Split Ordering
Recursive Split Ordering

0 → 4 → 2 → 6 → 1 → 5 → 3 → 7

1/4 1/2 3/4
Recursive Split Ordering

List entries sorted in order that allows recursive splitting. How?
Recursive Split Ordering
Recursive Split Ordering

LSB 0

0 → 4 → 2 → 6

LSB 1

1 → 5 → 3 → 7
Recursive Split Ordering

LSB 00  LSB 10  LSB 01  LSB 11

0 → 4 → 2 → 6 → 1 → 5 → 3 → 7

0 1 2 3
Split-Order

• If the table size is $2^i$,
  - Bucket $b$ contains keys $k$
    • $k = b \pmod{2^i}$
  - bucket index consists of key's $i$ LSBs
When Table Splits

- **Some keys stay**
  - \( b = k \mod (2^{i+1}) \)

- **Some move**
  - \( b + 2^i = k \mod (2^{i+1}) \)

- **Determined by \((i+1)^{st}\) bit**
  - Counting backwards
A Bit of Magic

Real keys:

0 4 2 6 1 5 3 7
A Bit of Magic

Real keys:

0  4  2  6  1  5  3  7

Real key 1 is in the 4th location

Split-order:

0  1  2  3  4  5  6  7
A Bit of Magic

Real keys:

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>2</th>
<th>6</th>
<th>1</th>
<th>5</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>100</td>
<td>010</td>
<td>110</td>
<td>001</td>
<td>101</td>
<td>011</td>
<td>111</td>
</tr>
</tbody>
</table>

Real key 1 is in 4\textsuperscript{th} location

Split-order:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>
A Bit of Magic

Real keys:

\[
\begin{array}{cccccccc}
000 & 100 & 010 & 110 & 001 & 101 & 011 & 111 \\
\end{array}
\]

Split-order:

\[
\begin{array}{cccccccc}
000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\
\end{array}
\]

Just reverse the order of the key bits
Split Ordered Hashing

Order according to reversed bits

000 001 010 011 100 101 110 111

0 → 4 → 2 → 6 → 1 → 5 → 3 → 7
Problem: how to remove a node pointed by 2 sources using CAS
Sentinel Nodes

Solution: use a Sentinel node for each bucket
Empirical Evaluation

- On a 30-processor Sun Enterprise 3000
- Lock-Free vs. Fine-Grained (D. Lea) Optimistic

- $10^6$ operations: 88% contains(), 10% add(), 2% remove()
Number of Fine-Grain Locks

ops/ns

threads

Lea, 64 locks
Lea, 128 locks
Lea, 32 locks
Lea, 16 locks
Lea, 8 locks
Lea, 256 locks
Lock-free vs Locks

The graph illustrates the performance comparison between lock-free and lock-based operations. The x-axis represents the number of threads, and the y-axis shows the number of operations per second (ops/ms). The graph compares two methods: "New" and "Lea". The "New" method shows a steady increase in performance with the number of threads, peaking around 50 threads before stabilizing. The "Lea" method, however, reaches a maximum at around 10 threads and then stabilizes, indicating a different behavior and performance characteristics.

This data suggests that lock-free methods can handle more threads efficiently, whereas lock-based methods might reach a saturation point at lower thread counts.
Coarse-Grained Simulation for Exascale Software Evaluation
The Problem: Exascale

• We don’t yet completely understand the incredibly complex design space in the exascale regime
  - Application scaling
  - Programming, communication, execution models
  - Data management
  - Fault tolerance at all levels
  - Power, performance, and cost tradeoffs for hardware technologies

• What we will end up with if we don’t understand:
  - A machine we can’t turn on (because the power: $$$)
  - A machine that turns itself off (because it fails a lot)
  - Extreme diminishing returns on machine investment for useful work (application performance)
SST/macro

http://sst.sandia.gov/about_sstmacro.html

• A coarse-grained communication and synchronization simulator
  - Network (structure, features, parameters)
  - Libraries (MPI, etc)
  - Application (skeletons)

• Built to scale
  - Lightweight threads (> 1M)
  - No computation in skeleton
SST/Macro Projects

**ASCR Execution Models**
- SST
  - System level models
- ACE
  - Node level emulation

**ASCR ExaCT: Combustion Codesign**

**SST/Macro**
- SST/Macro is a key tool in a number of exascale/co-design efforts.
- It brings key capabilities that work synergistically with other tools to form a complete picture.