Hashing and Natural Parallelism

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
Sequential Closed Hash Map

h(k) = k mod 4

2 Items
Add an Item

h(k) = k mod 4

3 Items
Add Another: Collision

4 Items

\( h(k) = k \mod 4 \)
More Collisions

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

5 Items
More Collisions

Problem: buckets becoming too long

5 Items

h(k) = k mod 4

Art of Multiprocessor Programming
Resizing

Grow the array

5 Items

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

Adjust hash function

h(k) = k mod 8

5 Items

Art of Multiprocessor Programming
Resizing

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

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

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

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

- 0 \rightarrow 16
- 1 \rightarrow 9
- 2 \rightarrow 7
- 3 \rightarrow 15
- 4 \rightarrow 4
- 5
- 6
- 7
Resizing

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

\[ h(7) = h(15) = 7 \mod 8 \]
Resizing

$h(k) = k \text{ mod } 8$

$h(15) = 7 \text{ mod } 8$
Fields

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

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

    ...  
```

Array of lock-free lists
public class SimpleHashSet {
    protected LockFreeList[] table;

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

    ...

    Initial size
public class SimpleHashSet {
    protected LockFreeList[] table;

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

    ...
public class SimpleHashSet {
    protected LockFreeList[] table;

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

    ...

    Initialization
Add Method

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

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

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

- Constant-time method calls require
  - Constant-length buckets
  - Table size proportional to set size
  - As set grows, must be able to resize
Set Method Mix

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

• Growing is important
• Shrinking not so much
When to Resize?

• Many reasonable policies. Here’s one.
• Pick a threshold on num of items in a bucket
• Global threshold
  – When $\geq \frac{1}{4}$ buckets exceed this value
• Bucket threshold
  – When any bucket exceeds this value
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 root reference didn’t change between resize decision and lock acquisition.

Acquire locks in ascending order.
Resize This

Allocate new super-sized table
Resize This

root

Art of Multiprocessor Programming
Striped Locks: each lock now associated with two buckets

Art of Multiprocessor Programming
Observations

• We grow the table, but not locks
  – Resizing lock array is tricky …

• We use sequential lists
  – Not LockFreeList lists
  – If we’re locking anyway, why pay?
Fine-Grained Hash Set

```java
public class FGHashSet {
    protected RangeLock[] lock;
    protected List[] table;
    public FGHashSet(int capacity) {
        table = new List[capacity];
        lock = new RangeLock[capacity];
        for (int i = 0; i < capacity; i++) {
            lock[i] = new RangeLock();
            table[i] = new LinkedList();
        }
    }
    ...}
```
Fine-Grained Hash Set

public class FGHashSet {
    protected RangeLock[] lock;
    protected List[] table;
    public FGHashSet(int capacity) {
        table = new List[capacity];
        lock = new RangeLock[capacity];
        for (int i = 0; i < capacity; i++) {
            lock[i] = new RangeLock();
            table[i] = new LinkedList();
        }
    }
}

Array of locks
Fine-Grained Hash Set

```java
public class FGHashSet {
    protected RangeLock[] lock;
    protected List[] table;
    public FGHashSet(int capacity) {
        table = new List[capacity];
        lock = new RangeLock[capacity];
        for (int i = 0; i < capacity; i++) {
            lock[i] = new RangeLock();
            table[i] = new LinkedList();
        }
    }
}
```

Array of buckets
Fine-Grained Hash Set

public class FGHashSet {
    protected RangeLock[] lock;
    protected List[] table;
    public FGHashSet(int capacity) {
        table = new List[capacity];
        lock = new RangeLock[capacity],
        for (int i = 0; i < capacity; i++) {
            lock[i] = new RangeLock();
            table[i] = new LinkedList();
        }
    }
}...

Initially same number of locks and buckets
The add() method

```java
public boolean add(Object key) {
    int keyHash = key.hashCode() % lock.length;
    synchronized (lock[keyHash]) {
        int tabHash = key.hashCode() % table.length;
        return table[tabHash].add(key);
    }
}
```
public boolean add(Object key) {
    int keyHash = key.hashCode() % lock.length;
    synchronized (lock[keyHash]) {
        int tabHash = key.hashCode() % table.length;
        return table[tabHash].add(key);
    }
}
The add() method

```java
public boolean add(Object key) {
    int keyHash = key.hashCode() % lock.length;
    synchronized (lock[keyHash]) {
        int tabHash = key.hashCode() % table.length;
        return table[tabHash].add(key);
    }
}
```

Acquire the lock
public boolean add(Object key) {
    int keyHash = key.hashCode() % lock.length;
    synchronized (lock[keyHash]) {
        int tabHash = key.hashCode() % table.length;
        return table[tabHash].add(key);
    }
}
The add() method

```java
public boolean add(Object key) {
    int keyHash = key.hashCode() % lock.length;
    synchronized (lock[keyHash]) {
        int tabHash = key.hashCode() % table.length;
        return table[tabHash].add(key);
    }
}
```
Another Locking Structure

- **add, remove, contains**
  - Lock table in *shared* mode

- **resize**
  - Locks table in *exclusive* mode
Read-Write Locks

```java
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```
Read/Write Locks

```java
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

Returns associated read lock
Read/Write Locks

public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}

Returns associated read lock
Returns associated write lock
Lock Safety Properties

• **Read lock:**
  – Locks out writers
  – Allows concurrent readers

• **Write lock**
  – Locks out writers
  – Locks out readers
Let's Try to Design a Read-Write Lock

• Read lock:
  – Locks out writers
  – Allows concurrent readers

• Write lock
  – Locks out writers
  – Locks out readers
Read/Write Lock

• Safety
  – If readers > 0 then writer == false
  – If writer == true then readers == 0

• Liveness?
  – Will a continual stream of readers ...
  – Lock out writers?
FIFO R/W Lock

- As soon as a writer requests a lock
- No more readers accepted
- Current readers “drain” from lock
- Writer gets in
ByteLock

• Readers-Writers lock
• Cache-aware
• Fast path for “slotted readers”
• Slower path for others
ByteLock Lock Record

Byte for each “slotted” reader

Writer id

Single cache line

W 0 0 0 0 0 R

Count of “unslotted” readers
Writer

Writer i

W=0

CAS

0 1 0 R=3
Writer

Writer $i$

Wait for readers to drain out

$W=i$ 0 0 1 0  R=3
Writer

Writer i

W=i 0 0 0 0 R=0

proceed
Slotted Reader Fast Path

Slotted Reader

W=0 0 0 1 0 R=3
Slotted Reader Fast Path

Slotted Reader

W=0 0 1 1 0 R=3
Slotted Reader Fast Path

Slotted Reader

W=0 0 1 1 0 R=3
Slotted Reader Fast Path
Slotted Reader Slow Path

Slotted Reader

W=i 0 1 1 0 R=3
Slotted Reader Slow Path

Slotted Reader

W=i 0 1 1 0 R=3
Slotted Reader Slow Path

Slotted Reader

W=0 0 0 1 0 R=3
Slotted Reader Slow Path

Slotted Reader

W=i 0 1 0 0 R=3
Unslotted Reader

W=i 0 1 0
R=4  
CAS
Slotted Reader Slow Path

Slotted Reader

W=i 0 1 1 0 R=3
Unslotted Reader

UnslottedReader

W=i 0 1 0 R=3 CAS
Slotted Reader Slow Path

Slotted Reader

W=i 0 1 0 0 R=3
The Story So Far

• Resizing is the hard part
• Fine-grained locks
  – Striped locks cover a range (not resized)
• Read/Write locks
  – FIFO property tricky
Stop The World Resizing

- Resizing stops all concurrent operations
- What about an incremental resize?
- Must avoid locking the table
- A lock-free table + incremental resizing?
Lock-Free Resizing Problem
Lock-Free Resizing Problem

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

We need a new idea...
Don’t move the items

- Move the buckets instead!
- Keep all items in a single, lock-free list
- Buckets are short-cuts into the list
Recursive Split Ordering
Recursive Split Ordering

0 → 4 ← 2 ← 6 → 1 ← 5 ← 3 → 7

1/2
Recursive Split Ordering
Recursive Split Ordering

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

0 → 4 → 2 → 6 → 1 → 5 → 3 → 7
Recursive Split Ordering

LSB 0

0 4 2 6

LSB 1

1 5 3 7

LSB = Least significant Bit
Recursive Split Ordering

LSB 00  LSBB 10  LSB 01  LSB 11

0 4 2 6 1 5 3 7
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

• Key must be accessible from both
  – Keys that will move must come later
A Bit of Magic

Real keys:

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

Real keys:

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

Split-order:
A Bit of Magic

Real keys:

```
0  4  2  6  1  5  3  7
000 100 010 110 001 101 011 111
```

Real key 1 is in 4th location

Split-order:

```
0  1  2  3  4  5  6  7
000 001 010 011 100 101 110 111
```
A Bit of Magic

Real keys:

000 100 010 110 001 101 011 111

Split-order:

000 001 010 011 100 101 110 111
A Bit of Magic

Real keys:

Split-order:

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

0 1 2 3
Bucket Relations

parent

child
Parent Always Provides a Short Cut
Sentinel Nodes

Problem: how to remove a node pointed by 2 sources using CAS
Solution: use a Sentinel node for each bucket
Sentinel vs Regular Keys

• Want sentinel key for \( i \) ordered
  – before all keys that hash to bucket \( i \)
  – after all keys that hash to bucket \((i-1)\)
Splitting a Bucket

• We can now split a bucket
• In a lock-free manner
• Using two CAS() calls ...
  – One to add the sentinel to the list
  – The other to point from the bucket to the sentinel
Initialization of Buckets

![Initialization of Buckets Diagram]
Initialization of Buckets

Need to initialize bucket 3 to split bucket 1
Adding 10

Must initialize bucket 2
Before adding 10
Recursive Initialization

To add 7 to the list

\[ 7 = 3 \mod 4 \]

\[ 7 = 1 \mod 2 \]

Could be \( \log n \) depth

But expected depth is constant

Must initialize bucket 1

Must initialize bucket 3
Lock-Free List

```c
int makeRegularKey(int key) {
    return reverse(key | 0x80000000);
}
int makeSentinelKey(int key) {
    return reverse(key);
}
```
Lock-Free List

int makeRegularKey(int key) {
    return reverse(key | 0x80000000);
}

int makeSentinelKey(int key) {
    return reverse(key);
}

Regular key: set high-order bit to 1 and reverse
Lock-Free List

int makeRegularKey(int key) {
    return reverse(key | 0x80000000);
}

int makeSentinelKey(int key) {
    return reverse(key);
}

Sentinel key: simply reverse (high-order bit is 0)
Main List

- Lock-Free List from earlier class
- With some minor variations
Lock-Free List

public class LockFreeList {
    public boolean add(Object object, int key) {...}
    public boolean remove(int k) {...}
    public boolean contains(int k) {...}
    public LockFreeList(LockFreeList parent, int key) {...};
}
Lock-Free List

```java
public class LockFreeList {

    public boolean add(Object object, int key) {...}
    public boolean remove(int k) {...}
    public boolean contains(int k) {...}
    public LockFreeList(LockFreeList parent, int key) {...};
}
```

Change: add takes key argument
Lock-Free List

Inserts sentinel with key if not already present...

```java
public class LockFreeList {
    public boolean add(Object object, int key) {...}
    public boolean remove(int k) {...}
    public boolean contains(int k) {...}
    public LockFreeList(LockFreeList parent, int key) {...};
}
```
Lock-Free List

... returns new list starting with sentinel (shares with parent)

```java
public class LockFreeList {
    public boolean add(Object object, int key) {...}
    public boolean remove(int k) {...}
    public boolean contains(int k) {...}
    public LockFreeList(LockFreeList parent, int key) {...};
}
```

returns new list starting with sentinel (shares with parent)
Split-Ordered Set: Fields

```java
public class SOSet {
    protected LockFreeList[] table;
    protected AtomicInteger tableSize;
    protected AtomicInteger setSize;

    public SOSet(int capacity) {
        table = new LockFreeList[capacity];
        table[0] = new LockFreeList();
        tableSize = new AtomicInteger(2);
        setSize = new AtomicInteger(0);
    }
}
```
Fields

public class SOSet {
    protected LockFreeList[] table;
    protected AtomicInteger tableSize;
    protected AtomicInteger setSize;

    public SOSet(int capacity) {
        table = new LockFreeList[capacity];
        table[0] = new LockFreeList();
        tableSize = new AtomicInteger(2);
        setSize = new AtomicInteger(0);
    }

    // For simplicity treat table as big array ...
}
Fields

In practice, want something that grows dynamically
Fields

public class SOSet {
    protected LockFreeList[] table;
    protected AtomicInteger tableSize;
    protected AtomicInteger setSize;

    public SOSet(int capacity) {
        table = new LockFreeList[capacity];
        table[0] = new LockFreeList();
        tableSize = new AtomicInteger(2);
        setSize = new AtomicInteger(0);
    }

    How much of table array are we actually using?
Fields

public class SOSet {
    protected LockFreeList[] table;
    protected AtomicInteger tableSize;
    protected AtomicInteger setSize;

    public SOSet(int capacity) {
        table = new LockFreeList[capacity];
        table[0] = new LockFreeList();
        tableSize = new AtomicInteger(2);
        setSize = new AtomicInteger(0);
    }

    // Track set size so we know when to resize
}
Fields

Initially use single bucket, and size is zero

```java
public class SOSet {
    protected LockFreeList[] table;
    protected AtomicInteger tableSize;
    protected AtomicInteger setSize;

    public SOSet(int capacity) {
        table = new LockFreeList[capacity];
        table[0] = new LockFreeList();
        tableSize = new AtomicInteger(1);
        setSize = new AtomicInteger(0);
    }
}
```
public boolean add(Object object) {
    int hash = object.hashCode();
    int bucket = hash % tableSize.get();
    int key = makeRegularKey(hash);
    LockFreeList list = getBucketList(bucket);
    if (!list.add(object, key))
        return false;
    resizeCheck();
    return true;
}
public boolean add(Object object) {
    int hash = object.hashCode();
    int bucket = hash % tableSize.get();
    int key = makeRegularKey(hash);
    LockFreeList list = getBucketList(bucket);
    if (!list.add(object, key))
        return false;
    resizeCheck();
    return true;
}
public boolean add(Object object) {
    int hash = object.hashCode();
    int bucket = hash % tableSize.get();
    int key = makeRegularKey(hash);
    LockFreeList list = getBucketList(bucket);
    if (!list.add(object, key))
        return false;
    resizeCheck();
    return true;
}
add()

```java
public boolean add(Object object) {
    int hash = object.hashCode();
    int bucket = hash % tableSize.get();
    int key = makeRegularKey(hash);
    LockFreeList list = getBucketList(bucket);
    if (!list.add(object, key))
        return false;
    resizeCheck();
    return true;
}
```

Get reference to bucket’s sentinel, initializing if necessary
```java
public boolean add(Object object) {
    int hash = object.hashCode();
    int bucket = hash % tableSize.get();
    int key = makeRegularKey(hash);
    LockFreeList list = getBucketList(bucket);
    if (!list.add(object, key))
        return false;
    resizeCheck();
    return true;
}
```

Call bucket’s add() method with reversed key.

```java
list.add(object, key)
```
add()

No change? We’re done.

```java
public boolean add(Object object) {
    int hash = object.hashCode();
    int bucket = hash % tableSize.get();
    int key = makeRegularKey(hash);
    LockFreeList list = getBucketList(bucket);
    if (!list.add(object, key))
        return false;
    resizeCheck();
    return true;
}
```
```java
public boolean add(Object object) {
    int hash = object.hashCode();
    int bucket = hash % tableSize.get();
    int key = makeRegularKey(hash);
    LockFreeList list = getBucketList(bucket);
    if (!list.add(object, key))
        return false;
    resizeCheck();
    return true;
}
```
Resize

• Divide set size by total number of buckets

• If quotient exceeds threshold
  – Double `tableSize` field
  – Up to fixed limit
Initialize Buckets

• Buckets originally null
• If you find one, initialize it
• Go to bucket’s parent
  – Earlier nearby bucket
  – Recursively initialize if necessary
• Constant expected work
Recall: Recursive Initialization

To add 7 to the list

7 = 3 mod 4

= 1 mod 2

expected depth is constant

Must initialize bucket 1

Must initialize bucket 3
Initialize Bucket

```java
void initializeBucket(int bucket) {
    int parent = getParent(bucket);
    if (table[parent] == null)
        initializeBucket(parent);
    int key = makeSentinelKey(bucket);
    LockFreeList list =
        new LockFreeList(table[parent],
                         key);
}
```
Initialize Bucket

```c
void initializeBucket(int bucket) {
    int parent = getParent(bucket);
    if (table[parent] == null)
        initializeBucket(parent);
    int key = makeSentinelKey(bucket);
    LockFreeList list =
        new LockFreeList(table[parent],
                         key);
}
```

Find parent, recursively initialize if needed
void initializeBucket(int bucket) {
    int parent = getParent(bucket);
    if (table[parent] == null)
        initializeBucket(parent);
    int key = makeSentinelKey(bucket);
    LockFreeList list =
        new LockFreeList(table[parent],
                         key);
}
Initialize Bucket

Insert sentinel if not present, and get back reference to rest of list

```java
void initializeBucket(int bucket) {
    int parent = getParent(bucket);
    if (table[parent] == null)
        initializeBucket(parent);
    int key = makeSentinelKey(bucket);
    LockFreeList list =
        new LockFreeList(table[parent],
                         key);
}
```
Correctness

- Linearizable concurrent set
- Theorem: $O(1)$ expected time
  - No more than $O(1)$ items expected between two sentinels on average
  - Lazy initialization causes at most $O(1)$ expected recursion depth in `initializeBucket()`
- Can eliminate use of sentinels
Closed (Chained) Hashing

**Advantages:**
- with \( N \) buckets, \( M \) items, Uniform \( h \)
- retains good performance as table density \( (M/N) \) increases → less resizing

**Disadvantages:**
- dynamic memory allocation
- bad cache behavior (no locality)

Oh, did we mention that cache behavior matters on a multicore?
Open Addressed Hashing

– Keep all items in an array
– One per bucket
– If you have collisions, find an empty bucket and use it
– Must know how to find items if they are outside their bucket
Linear Probing*

contains(x) – search linearly from h(x) to h(x) + H recorded in bucket.

*Attributed to Amdahl…
Linear Probing

\[ h(x) = z \]

\[ H = 6 \]

\textbf{add}(x) – put in first empty bucket, and update H.
Linear Probing

• Open address means $M \ll N$
• Expected items in bucket same as Chaining
• Expected distance till open slot:
\[
\frac{1}{2} \left( 1 + \left( \frac{1}{1 - \frac{M}{N}} \right)^2 \right)
\]

$M/N = 0.5 \Rightarrow \text{search 2.5 buckets}$
$M/N = 0.9 \Rightarrow \text{search 50 buckets}$
Linear Probing

• Advantages:
  – Good locality ➔ fewer cache misses

• Disadvantages:
  – As M/N increases more cache misses
    • searching 10s of unrelated buckets
    • “Clustering” of keys into neighboring buckets
  – As computation proceeds “Contamination” by deleted items ➔ more cache misses
But cycles can form

\textbf{Cuckoo Hashing}

\textbf{add} (x) – if \( h_1(x) \) and \( h_2(x) \) full evict \( y \) and move it to \( h_2(y) \neq h_2(x) \). Then place \( x \) in its place.
Cuckoo Hashing

• Advantages:
  – `contains(x)`: deterministic 2 buckets
  – No clustering or contamination

• Disadvantages:
  – 2 tables
  – $h_i(x)$ are complex
  – As $M/N$ increases $\Rightarrow$ relocation cycles
  – Above $M/N = 0.5$ Add() does not work!
Concurrent Cuckoo Hashing

- Need to either lock whole chain of displacements (see book)
- or have extra space to keep items as they are displaced step by step.
Hopscotch Hashing

- Single Array, Simple hash function
- Idea: define *neighborhood* of original bucket
- In neighborhood items found quickly
- Use sequences of displacements to move items into their neighborhood
Hopscotch Hashing

**contains**(x) – search in at most \( H \) buckets (the hop-range) based on hop-info bitmap. In practice pick \( H \) to be 32.
Hopscotch Hashing

add(x) – probe linearly to find open slot. Move the empty slot via sequence of displacements into the hop-range of h(x).
Hopscotch Hashing

- contains
  - wait-free, just look in neighborhood
Hopscotch Hashing

- **contains**
  - wait-free, just look in neighborhood

- **add**
  - expected distance same as in linear probing
Hopscotch Hashing

- **contains**
  - wait-free, just look in neighborhood

- **add**
  - Expected distance same as in linear probing

- **resize**
  - neighborhood full less likely as $H \to \log n$
  - one word hop-info bitmap, or use smaller $H$ and default to linear probing of bucket
Advantages

• Good locality and cache behavior
• As table density \((M/N)\) increases \(\rightarrow\) less resizing
• Move cost to \(\text{add()}\) from \(\text{contains}(x)\)
• Easy to parallelize
Recall: Concurrent Chained Hashing

Striped Locks

Lock for `add()` and unsuccessful `contains()`
Concurrent Simple Hopscotch

$h(x)$

contains() is wait-free
Concurrent Simple Hopscotch

\[ \text{add}(x) \] – lock bucket, mark empty slot using CAS, add \( x \) erasing mark
**Concurrent Simple Hopscotch**

\[ \text{add}(x) \] – lock bucket, mark empty slot using CAS, lock bucket and update timestamp of bucket being displaced before erasing old value
Concurrent Simple Hopscotch

Contains(x) – traverse using bitmap and if ts has not changed after traversal item not found. If ts changed, after a few tries traverse through all items.
Is performance dominated by cache behavior?

- Test on multicores and uniprocessors:
  - Sun 64 way Niagara II, and
  - Intel 3GHz Xeon

- Benchmarks pre-allocated memory to eliminate effects of memory management
Sequential SPARC Throughput
90% contain, 5% insert, 5% remove
Hopscotch_D
Hopscotch_ND
LinearProbing
Chained
Cuckoo

table density
ops /ms

with memory pre-allocated
Sequential SPARC High-Density: Throughput
90% contain, 5% insert, 5% remove

Hopscotch_D
Hopscotch_ND
LinearProbing
Chained

table density
ops /ms

Hopscotch_D
Hopscotch_ND
LinearProbing
Chained
Sequential CoreDuo; Throughput
90% contain, 5% insert, 5% remove

Hopscotch_D
Hopscotch_ND
LinearProbing
Chained
Cuckoo
Cuckoo stops here
Concurrent SPARC Throughput

90% density; 70% contain, 15% insert, 15% remove

- Hopscotch_D
- Chained_PRE
- Chained_MTM

with memory
pre-allocated
with allocation
Concurrent SPARC Throughput

90% density; Cache-Miss per UnSuccessful-Lookup

- Hopscotch_D
- Chained_PRE
- Chained_MTM

CPUs
miss / ops

0 / ops
0.5 / ops
1 / ops
1.5 / ops
2 / ops
2.5 / ops
3 / ops

1 8 16 24 32 40 48 56 64
Summary

• *Chained hash* with striped locking is simple and effective in many cases

• *Hopscotch* with striped locking great cache behavior

• If incremental resizing needed go for *split-ordered*
This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- **You are free:**
  - to Share — to copy, distribute and transmit the work
  - to Remix — to adapt the work
- **Under the following conditions:**
  - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - http://creativecommons.org/licenses/by-sa/3.0/.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.