Sequential Closed Hash Map

h(k) = k mod 4

2 Items

h(k) = k mod 4
Add an Item

3 Items

h(k) = k mod 4
Add Another: Collision

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

4 Items
More Collisions

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

5 Items

Art of Multiprocessor Programming
More Collisions

Problem: buckets becoming too long

h(k) = k mod 4

5 Items
Resizing

 Grow the array

5 Items

$h(k) = k \mod 4$
Resizing

Adjust hash function

5 Items

h(k) = k mod 8
Resizing

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

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

h(k) = k mod 8

h(4) = 4 mod 8
Resizing

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

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

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

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

Art of Multiprocessor Programming
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();
    }

    ...
Constructor

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

Art of Multiprocessor Programming
Add Method

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

Call bucket’s add() method
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

90% `Contains()`

1% `remove()`

9% `add()`

Growing is important

Shrinking less so
When to Resize?

Many reasonable policies

Pick a threshold on number 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:**
- Simple
- Hard to mess up

**Bad:**
- 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

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

Resize This

Art of Multiprocessor Programming

30
Observations

We grow the table, but not locks

Resizing lock array is tricky …

We use sequential lists

Not LockFreeList lists

Since we’re locking anyway
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();
        }
    }
} …
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();
        }
    }

    ...
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();
    }
  }
}

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);
    }
}
```
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

public interfaceReadWriteLock
{
    Lock readLock();
    Lock writeLock();
}

Read/Write Locks

Returns associated read lock
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}

Read/Write Locks

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
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 privileged *slotted readers*

Slower path for others
ByteLock Lock Record

- Byte for each “slotted” reader
- Writer id
- Single cache line
- Count of “unslotted” readers
Change writer from 0 to i

W=0

CAS

0 1 0  R=3
Writer

Wait for readers to drain …

W=i 0 0 1 0 R=3
Writer

Proceed when readers gone (via local spinning)

W=i 0 0 0 0 R=0
Writer

Release by resetting writer to 0

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

Slotted Reader

Change my slot from 0 …

\[ W=0 \ 0 \ 0 \ 0 \ 0 \ R=3 \]
Slotted Reader Fast Path

Slotted Reader

To 1

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

Is there a writer?

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

If writer found, reset byte to 0

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

Spin until no writer seen, then try again

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

Release by setting byte to 0

W=i 0 1 1 0 R=3

Slotted Reader
Slotted Reader Slow Path

Slotted Reader

Release by setting byte to 0

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

Increment unslotted readers count

W=i 0 1 0  

R=4

CAS
Slotted Reader Slow Path

Is there a writer?

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

If writer exists, decrement read counter, wait for writer to go away, then retry

W=i 0 1 0

R=3

CAS
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

The diagram illustrates a lock-free resizing problem with data nodes labeled 0 to 3. The nodes are interconnected as follows:

- Node 0 connects to node 4, which in turn connects to node 8.
- Node 1 connects to node 9.
- Node 2 connects to node 7, which in turn connects to node 15.

This structure represents a data structure where nodes are being resized and moved without locking, ensuring efficiency and concurrency in multiprocessor programming.
Lock-Free Resizing Problem

Need to extend table
Lock-Free Resizing Problem

Art of Multiprocessor Programming
Lock-Free Resizing Problem

We need a new idea…

Need to remove then add an 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 are short-cuts into the list
Recursive Split Ordering
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 → 0 → 4 → 2 → 6 → 1 → 5 → 3 → 7 → LSB 11
 LSB 10 → 1 → 5 → 3 → 7 → LSB 11
 LSB 01 → 0 → 4 → 2 → 6 → 1 → 5 → 3 → 7 → LSB 11
Split-Order

If the table size is $2^i$,

Bucket $b$ contains keys $k$

$b = k \pmod{2^i}$

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 buckets

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:

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:

```
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

0 → 4 → 2 → 6 → 1 → 5 → 3 → 7
Split Ordered Hashing

![Diagram of Split Ordered Hashing]

- **Parent**
- **Child**
Parent Always Provides a Short Cut
Sentinel Nodes

Problem: how to remove a node pointed by 2 sources using CAS
Sentinel Nodes

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

Need to initialize bucket 3 to split bucket 1
Adding 10

Must initialize bucket 2
Recursive Initialization

To add 7 to the list

Must initialize bucket 1

Must initialize bucket 3

7 = 3 mod 4

Worst-case log n depth
But expected constant depth

0 → 8 → 12 → 1 → 3

0
1
2
3

0
1
2
3

= 1 mod 2
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

```c
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 lectures

With minor variations
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) {...};
}
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

Change: add takes key argument
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) {...};
}
```
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);
    }
}
```
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 ...
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);
}

In practice, want something that grows dynamically

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

guard

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

Initially use single bucket, and size is zero
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;
}
```
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;
}
public boolean add(Object object) {
    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;
}
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 \textit{expected} work
Recall: Recursive Initialization

To add 7 to the list

\[ 7 = 3 \mod 4 \]

\[ 7 = 1 \mod 2 \]

Worst-case \( \log n \) depth

But expected constant depth

Must initialize bucket 1

Must initialize bucket 3
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);
}
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);
}
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);
}
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);
}
Split-Ordered Hash is Linearizable

**Theorem**

- $O(1)$ items *expected* between two sentinels
- Lazy: $O(1)$ *expected* recursion depth
- Can eliminate sentinels
Closed (Chained) Hashing

N buckets, M items, Uniform h

Advantages
- good performance as density (M/N) increases
- less resizing

Disadvantages
- Dynamic memory allocation
- Poor cache behavior from lack of locality
Open Addressed Hashing

All items in an array

One item for bucket

On collision, find and use nearby empty bucket

Must find items not in own bucket
contains(x) – search linearly from h(x) to h(x) + H recorded in bucket.

*Attributed to Amdahl...
Linear Probing

\[ h(x) = 3 \]

\[ x = 6 \]

add(x) – put in first empty bucket, and update H
Linear Probing

Expected # items in bucket same as chaining

Expected distance till open slot

$$\frac{1}{2} \left( 1 + \left( \frac{1}{1 - M/N} \right)^2 \right)$$

$M/N = 0.5 \Rightarrow \text{search } 2.5 \text{ buckets}$

$M/N = 0.9 \Rightarrow \text{search } 50 \text{ buckets}$
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
Contamination by deleted items ➔ more cache misses
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
hi(x) are complex
As M/N increases ➔ relocation cycles
Above M/N = 0.5 add() does not work!
Concurrent Cuckoo Hashing

Can lock 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

\( h(x) \)

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

\( u, w, v, z, r, s \)

1 1 0 1 0 0 1 1

\text{add}(x) – \text{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

add
Expected distance same as in linear probing

resize
neighborhood full less likely as $H \rightarrow \log n$
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 `add()` from `contains(x)`
- Easy to parallelize
Recall: Concurrent Chained Hashing

Striped Locks

Lock for `add()` and `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**

`add(x)` – lock bucket, mark empty slot using CAS, lock bucket and update timestamp of bucket being displaced before erasing old value.

![Diagram showing the process of adding a value to the hopscotch data structure.](image)

1. Lock bucket
2. Mark empty slot
3. Use CAS to update timestamp of bucket being displaced
4. Erase 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?

Parallel and sequential benchmarks

With and without memory pre-allocation
Sequential SPARC Throughput
90% contain, 5% insert, 5% remove

Hopscotch_D
Hopscotch_ND
LinearProbing
Chained
Cuckoo
Sequential SPARC High-Density; Throughput
90% contain, 5% insert, 5% remove

Hopscotch_D
Hopscotch_ND
LinearProbing
Chained

Table density
ops /ms

0.9 0.91 0.92 0.93 0.94 0.95 0.96 0.97 0.98 0.99
0
500
1000
1500
2000
2500
3000
3500
4000

ops /ms

0.9 0.91 0.92 0.93 0.94 0.95 0.96 0.97 0.98 0.99
0
500
1000
1500
2000
2500
3000
3500
4000

Table density

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

Hopscotch_D
Hopscotch_ND
LinearProbing
Chained
Cuckoo

table density
ops /ms

Cuckoo stops here
Concurrent SPARC Throughput
90% density; 70% contain, 15% insert, 15% remove

Hopscotch_D
Chained_PRE
Chained_MTM

CPUs
ops /ms

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

- Hopscotch_D
- Chained_PRE
- Chained_MTM
Summary

Chained Hash / striped locks
Simple and effective in many cases

Hopscotch / striped locks
Good cache behavior

Split-ordered
Good for incremental resizing
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.
Good locality and cache behavior

As table density \((M/N)\) increases \(\rightarrow\) less resizing

Move cost to \texttt{add()} from \texttt{contains(x)}

Easy to parallelize
As computation proceeds, contamination by deleted items leads to more cache misses.

For $M/N = 0.5$, search 2.5 buckets.

For $M/N = 0.9$, search 50 buckets.

Must find items not in own bucket upon eviction.

Dynamic memory allocation.

Poor cache behavior from lack of locality.

Which properties are important?

Keys that will move must come later.