İTÜ

Hashing and Natural Parallism

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

Linked Lists

İTÜ

- We looked at a number of ways to make highly-concurrent list-based Sets:
  - Fine-grained locks
  - Optimistic synchronization
  - Lazy synchronization
  - Lock-free synchronization
- What's missing?

Linear-Time Set Methods

İTÜ

- Problem is
  - add(), remove(), contains()
  - Take time linear in set size
- We want
  - Constant-time methods
  - (at least, on average)

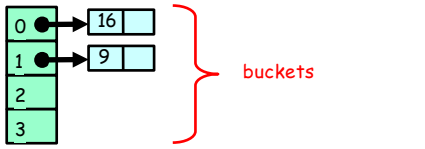
Hashing

İTÜ

- Hash function
  - h: items → integers
- Uniformly distributed
  - Different item most likely have different hash values
- Java hashCode() method

Sequential Hash Map

İTÜ

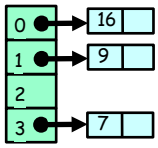


2 Items

h(k) = k mod 4

Add an Item

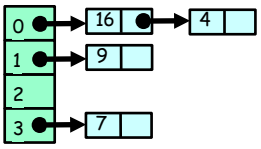
İTÜ



3 Items

h(k) = k mod 4

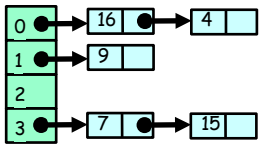
Add Another: Collision



4 Items

h(k) = k mod 4

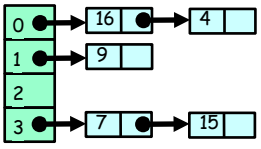
More Collisions



5 Items

h(k) = k mod 4

More Collisions

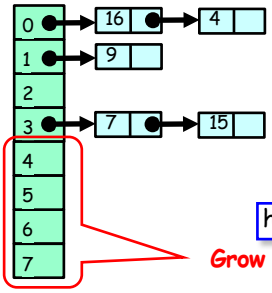


Problem:
buckets getting too long

5 Items

h(k) = k mod 4

Resizing

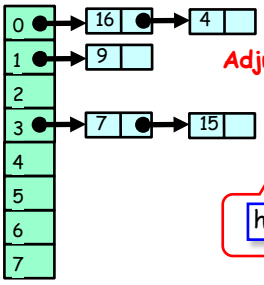


5 Items

h(k) = k mod 8

Grow the array

Resizing

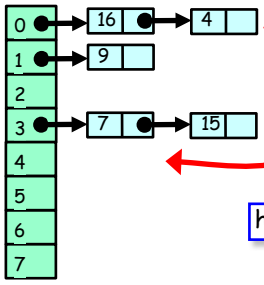


Adjust hash function

5 Items

h(k) = k mod 8

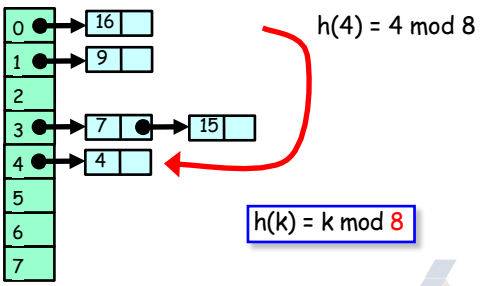
Resizing



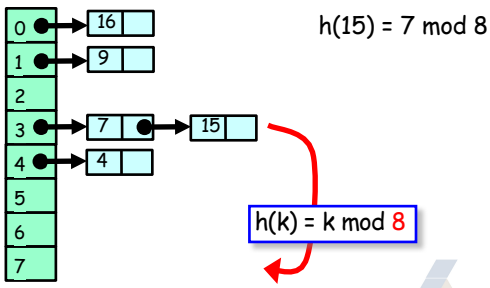
h(4) = 0 mod 8

h(k) = k mod 8

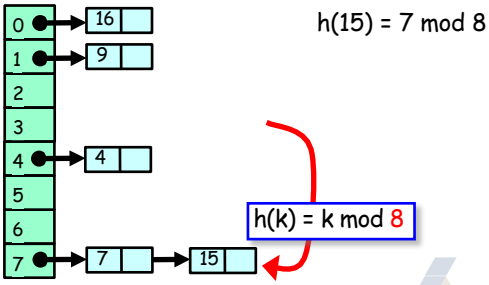
Resizing



Resizing



Resizing



Hash Sets

- Implement a Set object
  - Collection of items, no duplicates
  - add(), remove(), contains() methods
  - You know the drill …

Simple Hash Set

```
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();

  }
…
```


Add Method

```
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

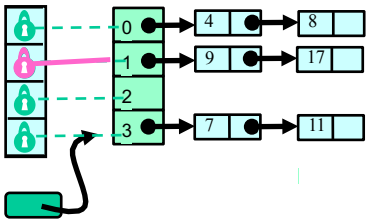
When to Resize?

- Many reasonable policies. Here's one.
- Pick a threshold on num of items in a bucket
- Global threshold
  - When ≥ ¼ 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

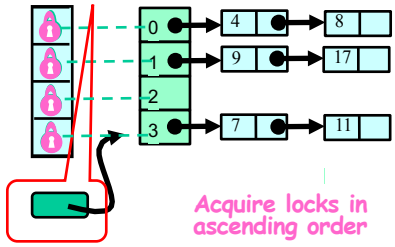
Fine-grained Locking



Each lock associated with one bucket

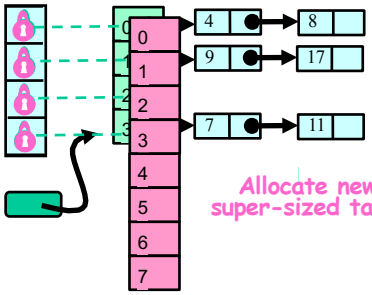
Resize This

Make sure table reference didn't change between resize decision and lock acquisition



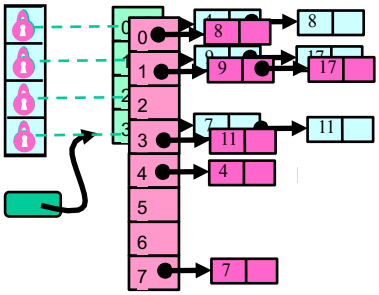
Acquire locks in ascending order

Resize This



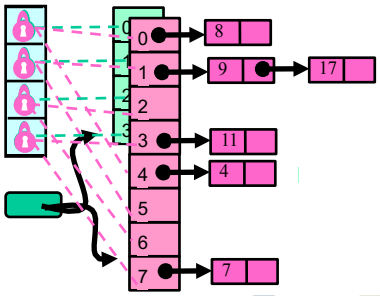
Allocate new super-sized table

Resize This



Resize This

Striped Locks: each lock now associated with two buckets



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

```
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();
  }} …
```


The add() method

```
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);

 }
}
```


Fine-Grained Locking

```
private void resize(int depth,List[] oldTab) {

 synchronized (lock[depth]) {

  if (oldTab == this.table){

   int next = depth + 1;

   if (next < lock.length)
    resize (next, oldTab);
   else
    sequentialResize();

 }}}
```


Fine-Grained Locking

```
private void resize(int depth,
                    List[] oldTab) {
 synchronized (lock[depth]) {
  if (oldTab == this.table){
   int next = depth + 1;
   if (next < lock.length)
    resize (next, oldTab);
   else
    sequentialResize();
 }}}
```

resize() calls resize(0,this.table)

Fine-Grained Locking

```
private void resize(int depth,
                    List[] oldTab) {
 synchronized (lock[depth]) {
  if (oldTab == this.table){
   int next = depth + 1;
   if (next < lock.length)
    resize (next, oldTab);
   else
    sequentialResize();
 }}}
```

Acquire next lock

Fine-Grained Locking

```
private void resize(int depth,
                    List[] oldTab) {
 synchronized (lock[depth]) {
  if (oldTab == this.table) {
   int next = depth + 1;
   if (next < lock.length)
    resize (next, oldTab);
   else
```

Check that no one else has resized

Fine-Grained Locking

İTÜ

Recursively acquire next lock

```
synchronized (lock[depth]) {
 if (oldTab == this.table){
  int next = depth + 1;
  if (next < lock.length)
   resize (next, oldTab);
  else
   sequentialResize();
}}}
```


Fine-Grained Locking

İTÜ

Locks acquired, do the work

```
synchronized (lock[depth]) {
 if (oldTab == this.table){
  int next = depth + 1;
  if (next < lock.length)
   resize (next, oldTab);
  else
   sequentialResize();
}}}
```


Fine-Grained Locks

İTÜ

- We can resize the table
- But not the locks
- Debatable whether method calls are constant-time in presence of contention …

Insight

İTÜ

- The contains() method
  - Does not modify any fields
  - Why should concurrent contains() calls conflict?

Read/Write Locks

İTÜ

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


Lock Safety Properties

İTÜ

- No thread may acquire the write lock
  - while any thread holds the write lock
  - or the read lock.
- No thread may acquire the read lock
  - while any thread holds the write lock.
- Concurrent read locks OK

Read/Write Lock

- Satisfies safety properties
  - If readers > 0 then writer == false
  - If writer = true then readers == 0
- Liveness?
  - Lots of readers …
  - Writers locked out?

FIFO R/W Lock

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

The Story So Far

- Resizing the hash table is the hard part
- Fine-grained locks
  - Striped locks cover a range (not resized)
- Read/Write locks
  - FIFO property tricky

Optimistic Synchronization

- If the contains() method
  - Scans without locking
- If it finds the key
  - OK to return true
  - Actually requires a proof ….
- What if it doesn't find the key?

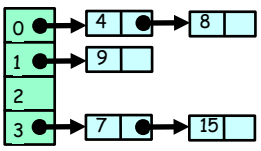
Optimistic Synchronization

- 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

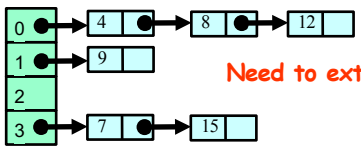
Stop The World Resizing

- 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

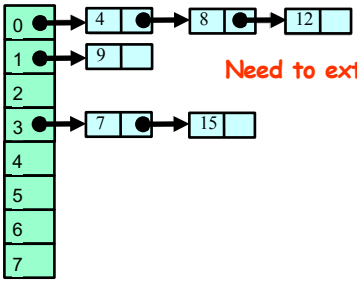


Lock-Free Resizing Problem



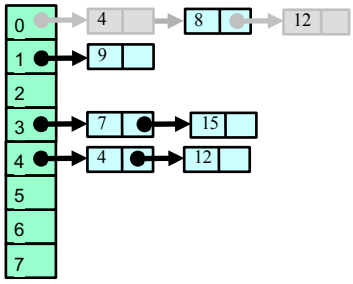
Need to extend table

Lock-Free Resizing Problem

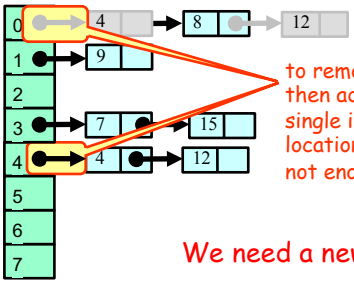


Need to extend table

Lock-Free Resizing Problem



Lock-Free Resizing Problem

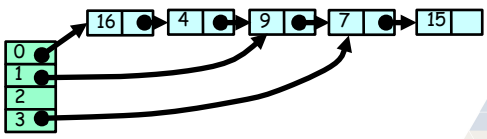


to remove and then add even a single item single location CAS not enough

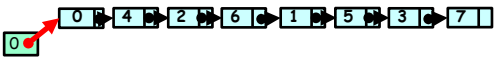
We need a new idea...

Don't move the items

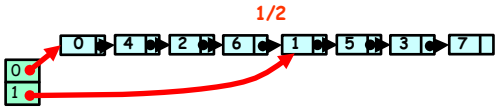
- Move the buckets instead
- Keep all items in a single lock-free list
- Buckets become "shortcut pointers" into the list



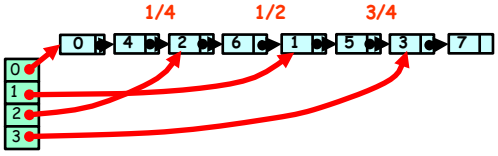
Recursive Split Ordering



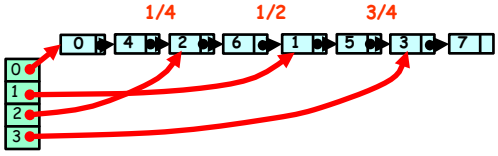
Recursive Split Ordering



Recursive Split Ordering

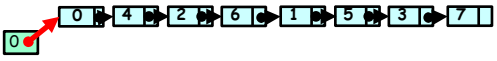


Recursive Split Ordering

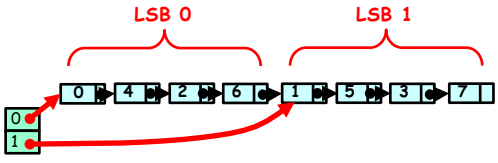


List entries sorted in order that allows recursive splitting. How?

Recursive Split Ordering

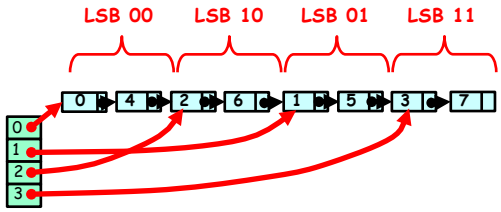


Recursive Split Ordering



LSB = Least significant Bit

Recursive Split Ordering



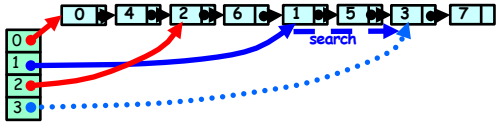
Split-Order

- 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

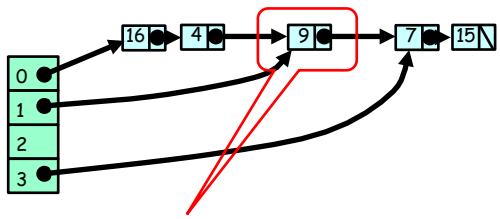
When Table Splits

- Some keys stay
  - $b = k \bmod(2^{i+1})$
- Some move
  - $b+2^i = k \bmod(2^{i+1})$
- Determined by $(i+1)^{st}$ bit
  - Counting backwards
- Key must be accessible from both
  - Keys that will move must come later

Parent Always Provides a Short Cut

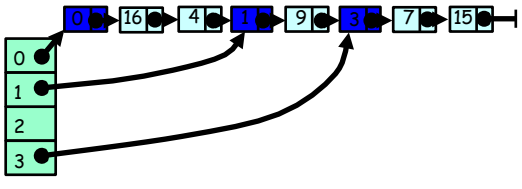


Sentinel Nodes



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

Sentinel Nodes



Solution: use a Sentinel node for each bucket

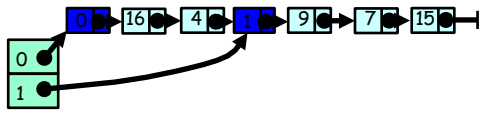
Sentinel vs Regular Keys

- Want sentinel key for i ordered
  - before all keys that hash to bucket i
  - after all keys that hash to bucket (i-1)

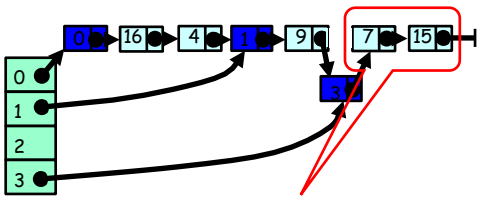
Splitting a Bucket

- We can now split a bucket
- In a lock-free manner
- Using two CAS() calls ...

Initialization of Buckets

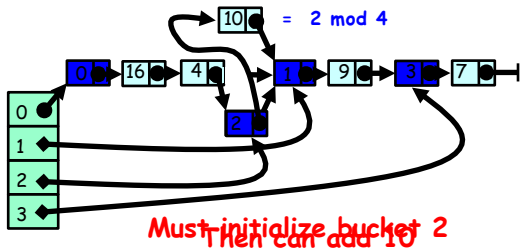


Initialization of Buckets



Now 3 points to sentinel – bucket has been split

Adding 10



Must initialize bucket 2
Then can add 10

Main List

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

Split-Ordered Set: 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);
  }
```


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();
    ta
    se
  }
```

For simplicity treat table as big array …

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();
    ta
    se
  }
```

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();
    ta
    se
  }
```

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();
    ta
    se
  }
```

Track set size so we know when to resize

Fields

Initially use 1 bucket and size is zero

```
  protected AtomicInteger setSize;

  public SOSet(int capacity) {
    table = new LockFreeList[capacity];
    table[0] = new LockFreeList();
    tableSize = new AtomicInteger(1);
    setSize = new AtomicInteger(0);
  }
```

Add() Method

```
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;
}
```

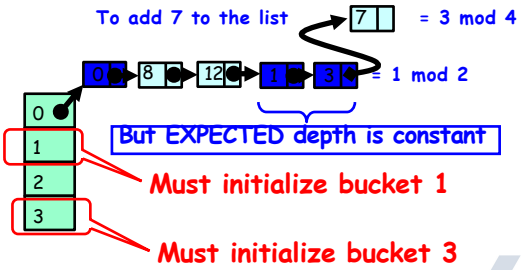

Resize

- 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



Initialize Bucket

```
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

```
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

Initialize Bucket

```
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);
 }
```

Prepare key for new sentinel

Initialize Bucket

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

```
 initializeBucket(parent);
int key = makeSentinelKey(bucket);
LockFreeList list =
  new LockFreeList(table[parent],
                   key);
 }
```

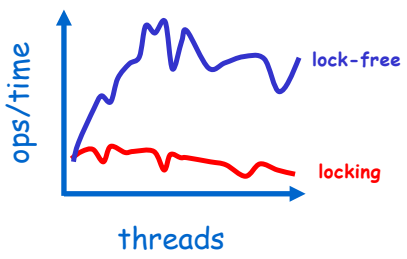

Correctness

- Linearizable concurrent set implementation
- Theorem: O(1) expected time
  - No more than O(1) items expected between two dummy nodes on average
  - Lazy initialization causes at most O(1) expected recursion depth in initializeBucket()

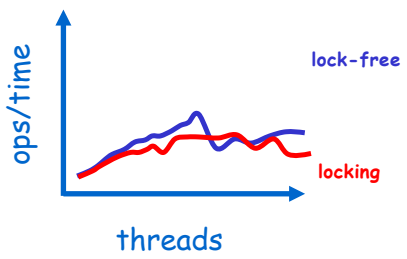
Empirical Evaluation

- On a 30-processor Sun Enterprise 3000
- Lock-Free vs. fine-grained (Lea) optimistic
- In a non-multiprogrammed environment
- 106 operations: 88% contains(), 10% add(), 2% remove()

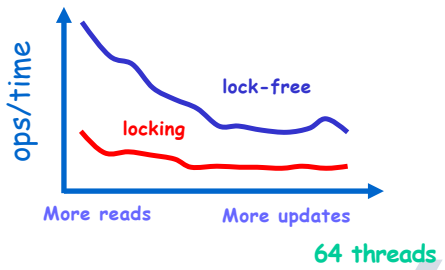
Work = 0



Work = 500



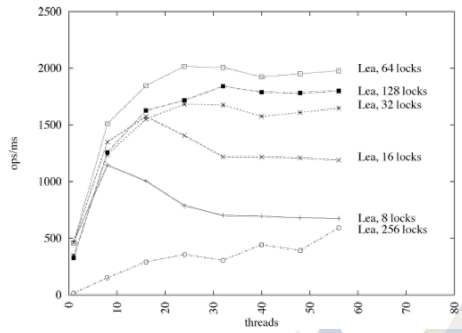
Varying The Mix



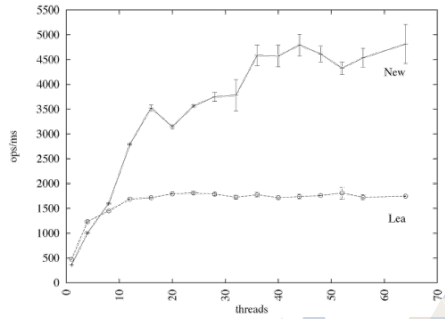
Additional Performance

- The effects of the choice of locking granularity
- The effects of bucket size

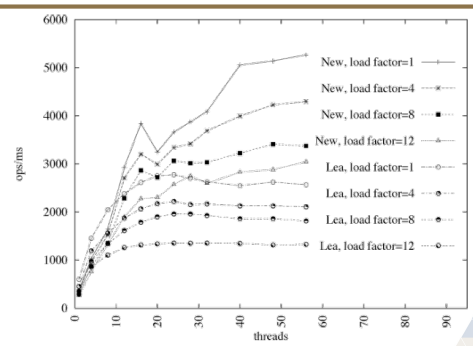
Number of Fine-Grain Locks



Lock-free vs Locks

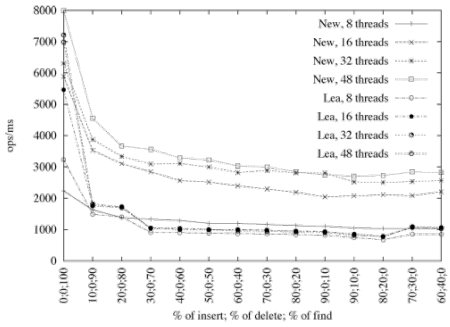


Hash Table Load Factor



(load factor = nodes per bucket)

Varying Operations



Summary

- Lock-based
  - Fine-grained
  - Read/write locks
  - Optimistic
- Lock-free
  - Builds on lock-free list