A Lock-Free Wait-Free Hash Table

UNSHACKLE THE
POWER OF JAVA

NETWORK ATTACHED PROCESSING
FROM AZUL SYSTEMS

Dr. Cliff Click
Distinguished Engineer

Hash Tables

- Constant-Time Key-Value Mapping
- Fast arbitrary function
- Extendable, defined at runtime
- Used for symbol tables, DB caching, network access, url caching, web content, etc
- Crucial for Large Business Applications
  - > 1MLOC
- Used in Very heavily multi-threaded apps
  - > 1000 threads

Popular Java Implementations

- Java's HashTable
  - Single threaded; scaling bottleneck
- HashMap
  - Faster but NOT multi-thread safe
- java.util.concurrent.HashMap
  - Striped internal locks; 16-way the default
- Azul, IBM, Sun sell machines >100cpus
- Azul has customers using all cpus in same app
- Becomes a scaling bottleneck!

A Wait-Free (Lock-Free) Hash Table

- No locks, even during table resize
  - No CAS spin-loops
- Requires CAS, LL/SC or other atomic-update
- Wait-free property requires CAS not fail spuriously
  - Or at least limited to finite spurious failures
  - Reason for failure dictates next action

A Faster Hash Table

- Tied with j.u.c for 99% reads < 32 cpus
- Faster with more cpus (3.5x faster)
  - Even with high striping levels
  - j.u.c with 1024 stripes still 2x slower
- Much faster for 95% reads (20x faster)
- Scales well up to 768 cpus, 75% reads
  - Approaches hardware bandwidth limits
- Scales up to 400 cpus, 50% reads

Agenda

- Motivation
- **“Uninteresting” Hash Table Details**
- State-Based Reasoning
- Resize
- Performance
- Q&A

Some “Uninteresting” Details

- Hashtable: A collection of Key/Value Pairs
- Works with any collection
- Scaling, locking, bottlenecks of the collection management responsibility of that collection
- Must be fast or O(1) effects kill you
- Must be cache-aware
- I'll present a sample Java solution
  - But other solutions can work, make sense

"Uninteresting" Details

- Closed Power-of-2 Hash Table
  - Reprobe on collision
  - Stride-1 reprobe: better cache behavior
- Key & Value on same cache line
- Hash memoized
  - Should be same cache line as K + V
  - But hard to do in pure Java
- No allocation on get() or put()
- Auto-Resize

“Uninteresting” Details

- Example get() work:

```
idx = hash = key.hashCode();
while( true ) {         // reprobing loop
   idx &= (size-1);     // limit idx to table size
   k = get_key(idx);    // start cache miss early
   h = get_hash(idx);   // memoized hash
   if( k == key || (h == hash && key.equals(k)) )
      return get_val(idx);// return matching value
   if( k == null ) return null;
   idx++;               // reprobe
}
```

“Uninteresting” Details

- Could use prime table + MOD
  - Better hash spread, fewer reprobes
  - But MOD is 30x slower than AND
- Could use open table
  - put() requires allocation
  - Follow 'next' pointer instead of reprobe
  - Each 'next' is a cache miss
  - Lousy hash -> linked-list traversal
- Could put Key/Value/Hash on same cache line
- Other variants possible, interesting

Agenda

- Motivation
- “Uninteresting” Hash Table Details
- **State-Based Reasoning**
- Resize
- Performance
- Q&A

Ordering and Correctness

- How to show table mods correct?
  - put, putIfAbsent, change, delete, etc.
- Prove via: fencing, memory model, load/store ordering, “happens-before”?
- Instead prove* via state machine
- Define all possible {Key,Value} states
- Define Transitions, State Machine
- Show all states “legal”

*Warning: hand-wavy proof follows

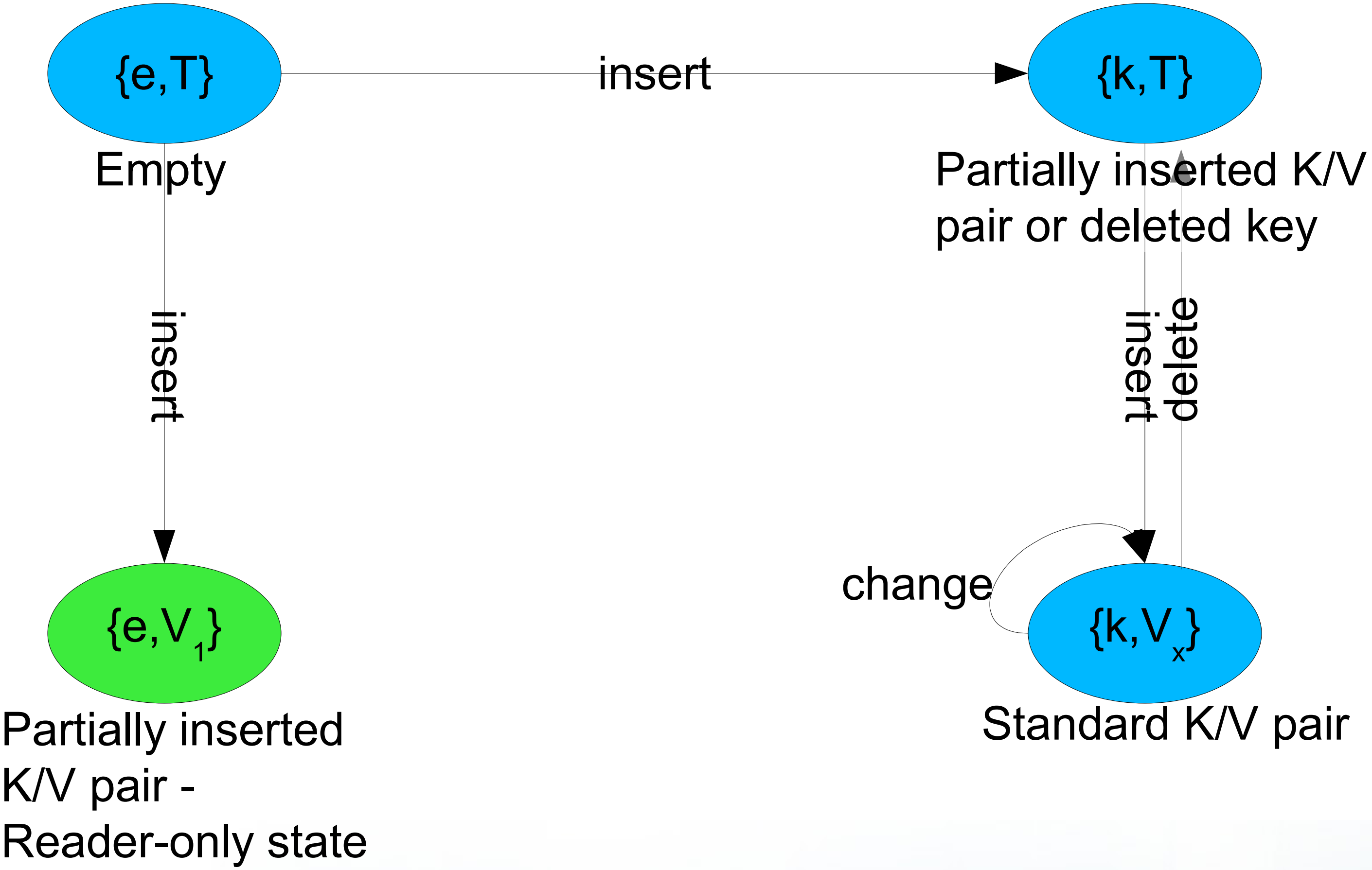
State-Based Reasoning

- Define all {Key,Value} states and transitions
- Don't Care about memory ordering:
  - get() can read Key, Value in any order
  - put() can change Key, Value in any order
  - put() must use CAS to change Key or Value
    - But not double-CAS
- No fencing required for correctness!
  - (sometimes stronger guarantees are wanted and will need fencing)
- Proof is simple!

Valid States

- A Key slot is:
  - e – empty
  - k – some Key; can never change again
- A Value slot is:
  - T – tombstone, empty
  - V_1, V_2– some Values
- A state is a {Key,Value} pair
- Initialize all pairs to empty
  - Handy to represent empty as null

State Machine



{e,T}

Empty

insert

{k,T}

Partially inserted K/V pair or deleted key

insert

{e,V_1}

Partially inserted K/V pair - Reader-only state

insert

delete

change

{k,V_x}

Standard K/V pair

Some Things to Notice

- Once a Key is set, it never changes
  - No chance of returning Value for wrong Key
  - Means Keys leak; table fills up with dead Keys
    - Fix in a few slides...
- No ordering guarantees provided!
  - Bring Your Own Ordering/Synchronization
- Weird {e,V} state meaningful but uninteresting
  - Means reader got an empty key and so missed
    - But possibly prefetched wrong Value

Some Things to Notice

- There is no machine-wide coherent State!
- Nobody guaranteed to read the same State
  - Except on the same CPU with no other writers
- No need for it either
- Consider degenerate case of a single Key
- Same guarantees as:
  - single shared global variable
  - many readers & writers, no synchronization
  - i.e., darned little

A Slightly Stronger Guarantee

- Probably want “happens-before” on Values
  - java.util.concurrent provides this
- Similar to declaring that shared global 'volatile'
- Things written into a Value before put()
  - Are guaranteed to be seen after a get()
- Requires st/st fence before CAS'ing Value
  - “free” on Sparc, X86
- Requires ld/ld fence after loading Value
  - “free” on Azul

Agenda

- Motivation
- “Uninteresting” Hash Table Details
- State-Based Reasoning
- **Resize**
- Performance
- Q&A

Resizing The Table

- Need to resize if table gets full
- Or just re-probing too often
- Resize copies live K/V pairs
  - Doubles as cleanup of dead Keys
  - Resize (“cleanse”) after any delete
  - Throttled, once per GC cycle is plenty often
- Alas, need fencing, 'happens before'
- Hard bit for concurrent resize & put():
  - Must not drop the last update to old table

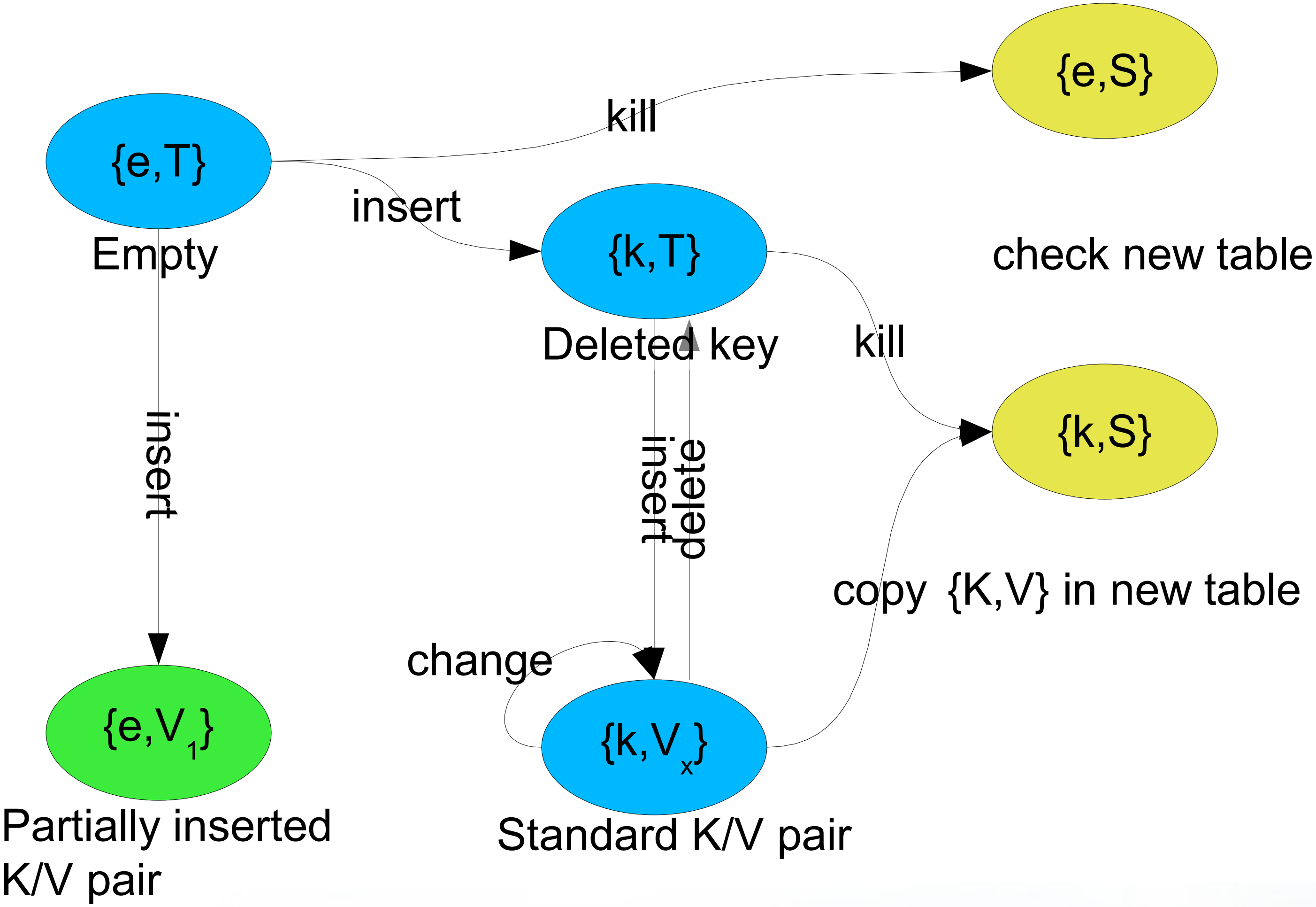
Resizing

- Expand State Machine
- Side-effect: mid-resize is a valid State
- Means resize is:
  - Concurrent – readers can help, or just read&go
  - Parallel – all can help
  - Incremental – partial copy is OK
- Pay an extra indirection while resize in progress
  - So want to finish the job eventually
- Stacked partial resizes OK, expected

New Valid States

- A Key slot is:
  - e – empty
  - k – some unchanging Key
- A Value slot is:
  - T – tombstone/empty
  - V_x – some Values
  - S – sentinel, not any valid Value
  - T',V' – primed versions of T & V
    - Old things copied into the new table
    - “2-phase commit”

State Machine



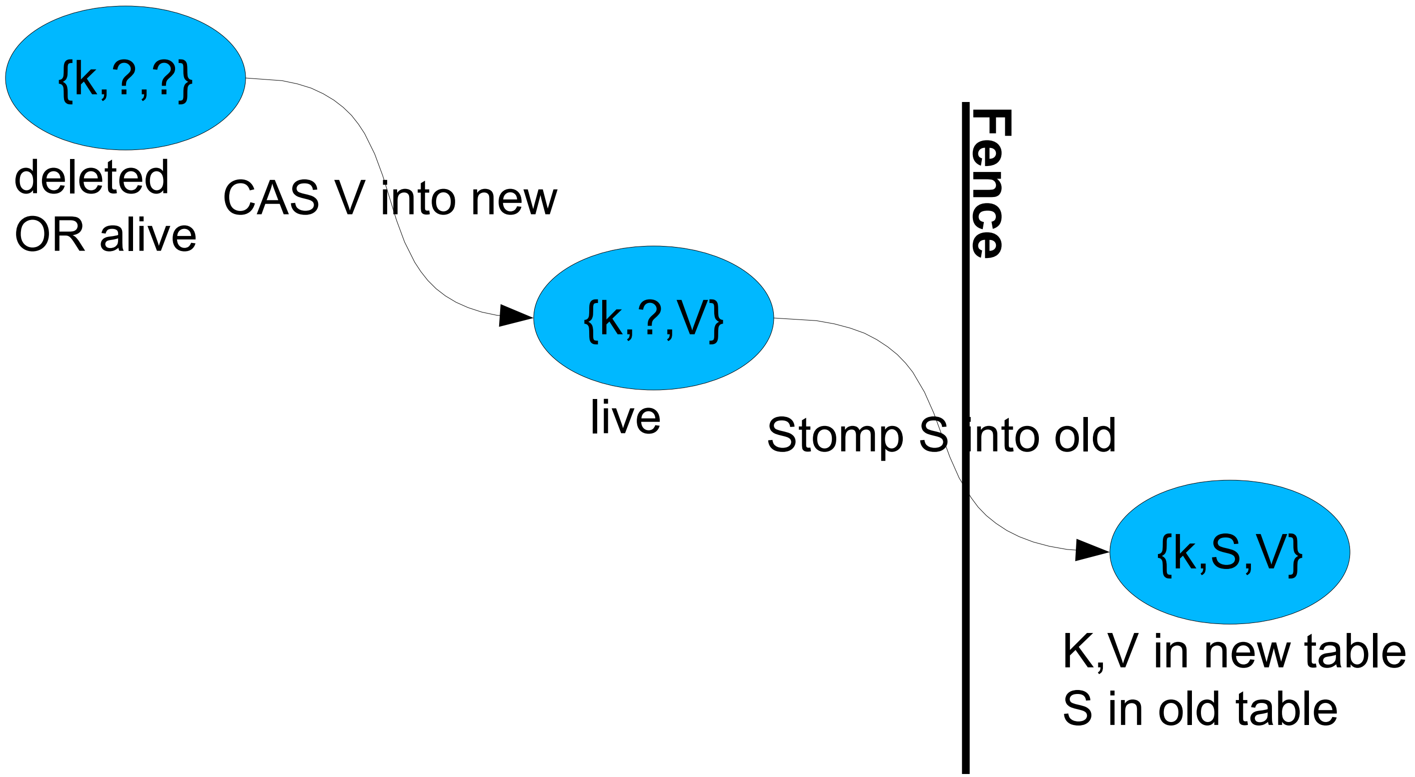
Resizing

- Copying K/V pairs is independent of get/put
- Many heuristics to choose from:
  - All touching threads, only writers, unrelated background thread(s), etc
- get() works on the old table
  - Unless see a sentinel
- put() or other mod *must* use new table
- Must check for new table every time
  - Late writes to old table 'happens before' resize

Resizing – put(K,V) while copy

- put() in new table, same as before
- Old Value will be overwritten, no need to read
- Fence!
- Store (not CAS) 'S' into old table
  - Stomps over old table
  - No longer care for what was there
- State Machine may help you visualize...
- New State includes both tables:
  - {Key, OldVal, NewVal}

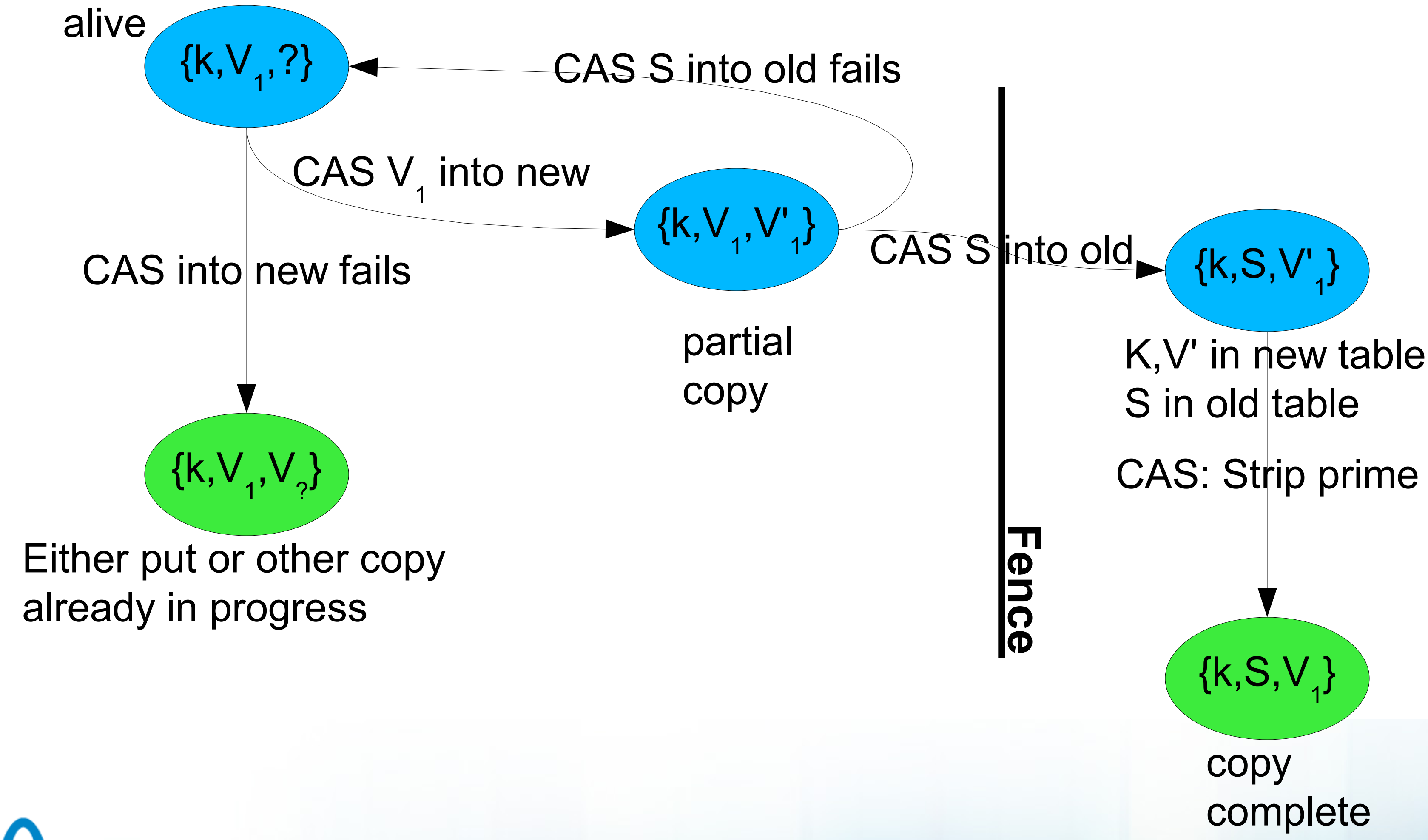
State Machine: put(K,V) while copy



Resizing – Normal Copy

- 'get()' thread or helper thread
- Must be sure to copy late-arriving old-table write
- Attempt to copy atomically
  - May fail & copy does not make progress
  - But old, new tables not damaged
- Prime allows 2-phase commit
  - Prime'd values copied from old
  - Non-prime is recent put()
    - “happens after” any prime'd value
- State Machine again...

State Machine: Copy One Pair



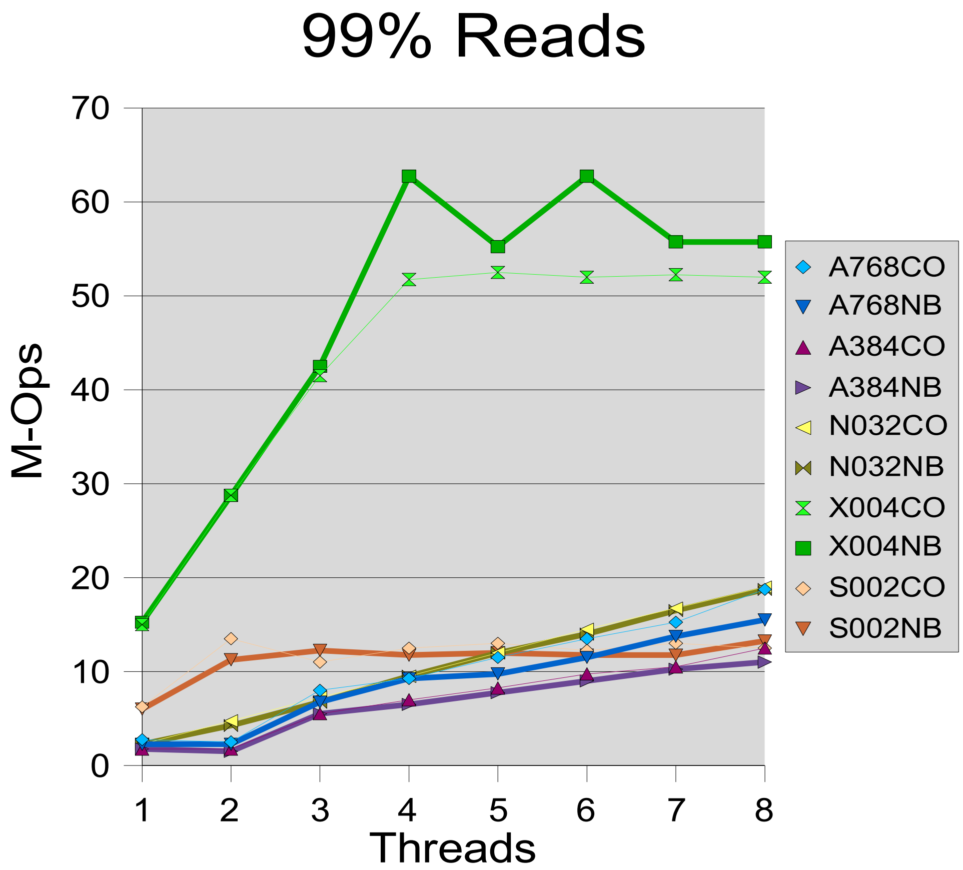
Some Things to Notice

- Old value could be V or T
  - or V' or T' (if nested resize in progress)
  - For old T, just CAS tombstone to S
  - no need to insert tombstone in new table
- Skip copy if new Value is not prime'd
  - Means recent put() overwrote any old Value
- If CAS into new fails
  - Means either put() or other copy in progress
  - So this copy can quit

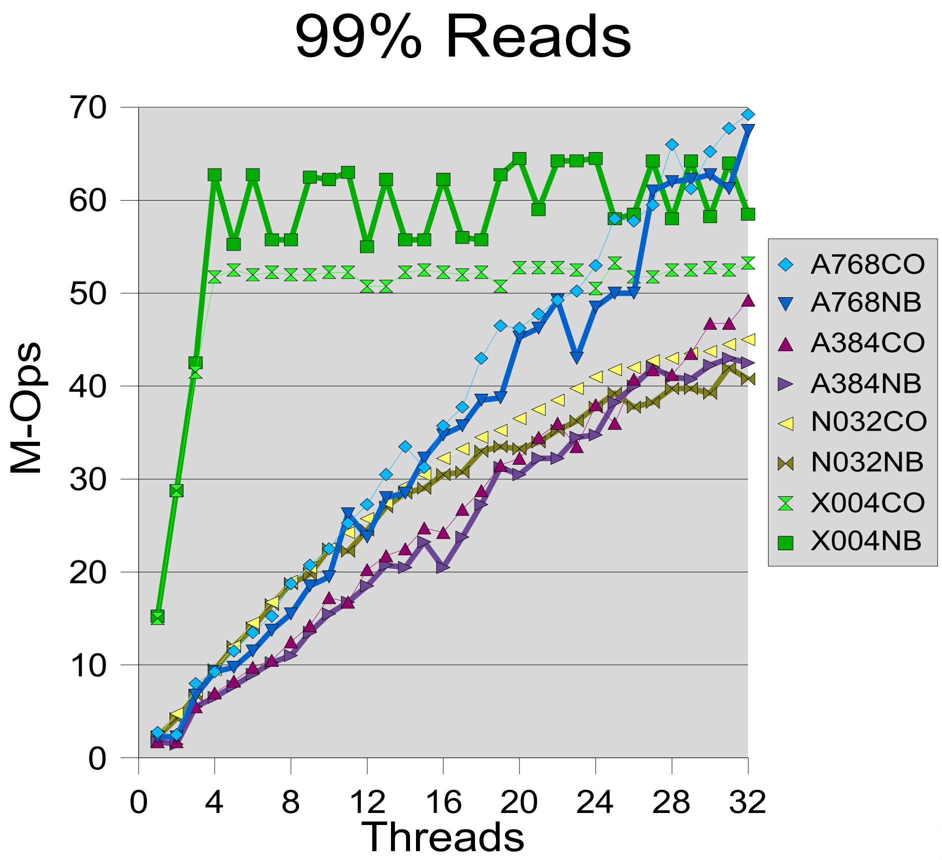
Agenda

- Motivation
- “Uninteresting” Hash Table Details
- State-Based Reasoning
- Resize
- **Performance**
- Q&A

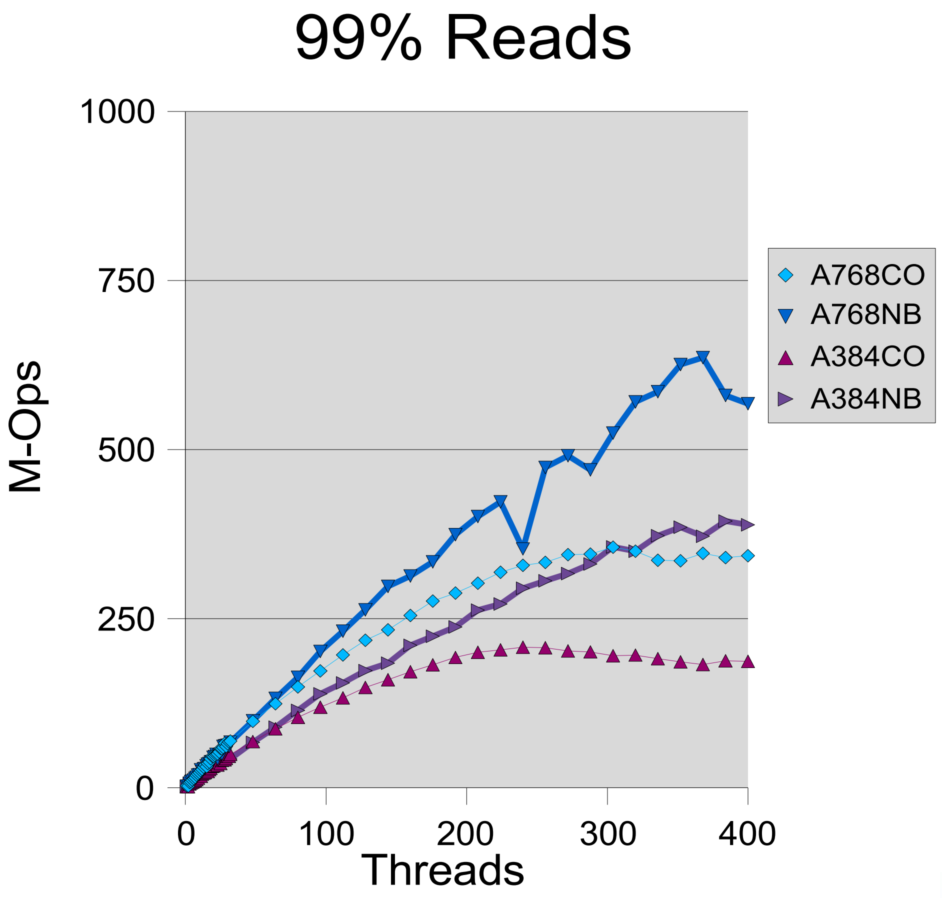
99% Reads


99% Reads
M-Ops
0
10
20
30
40
50
60
70
Threads
1
2
3
4
5
6
7
8
A768CO
A768NB
A384CO
A384NB
N032CO
N032NB
X004CO
X004NB
S002CO
S002NB

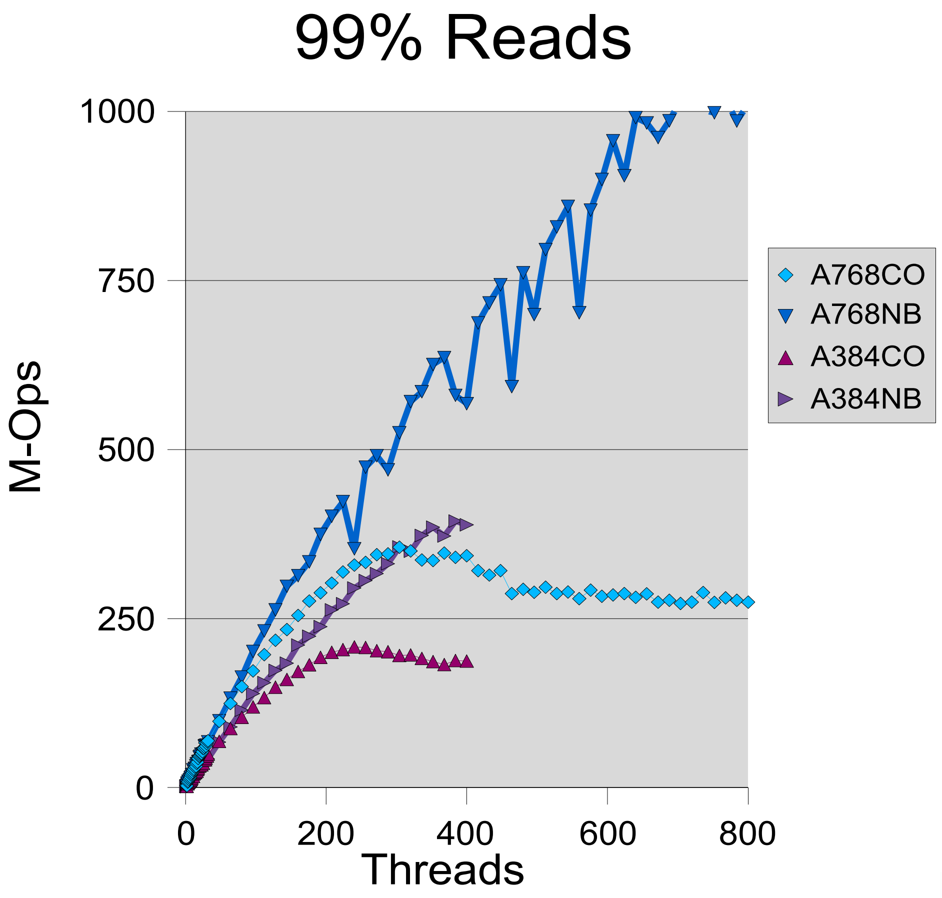
99% Reads


99% Reads
M-Ops
70
60
50
40
30
20
10
0
0
4
8
12
16
20
24
28
32
Threads
A768CO
A768NB
A384CO
A384NB
N032CO
N032NB
X004CO
X004NB

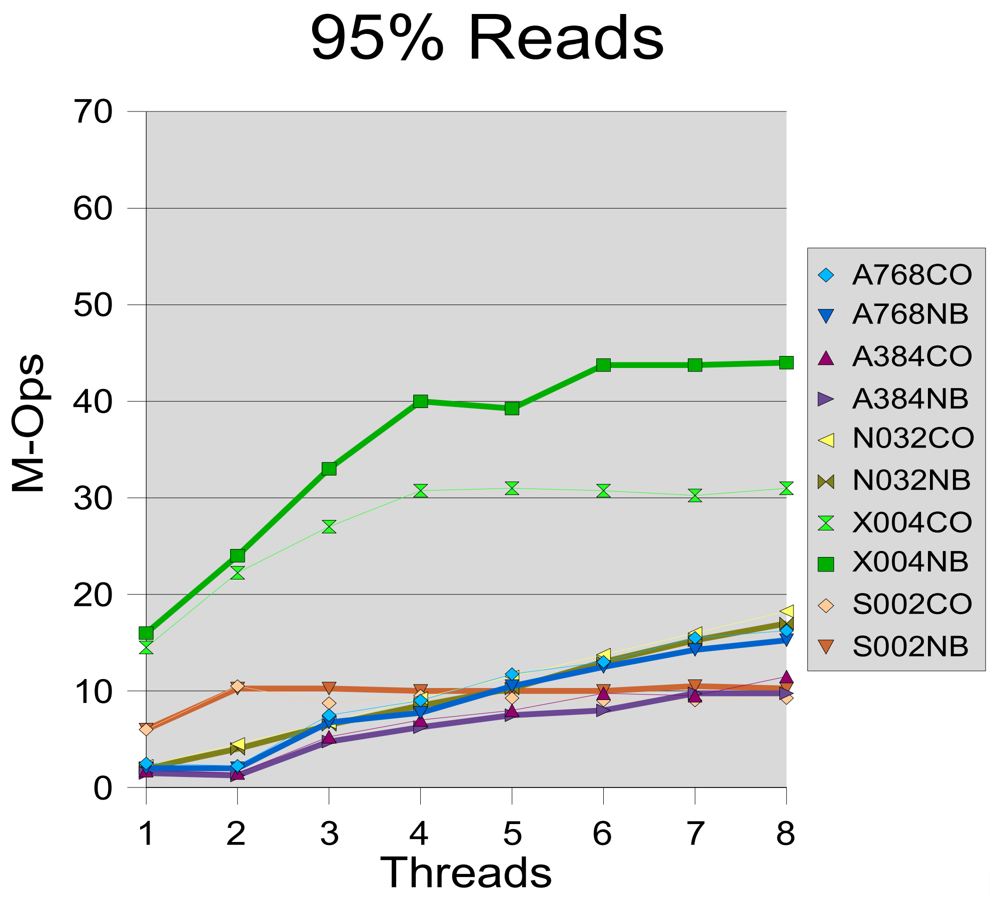
99% Reads


99% Reads
M-Ops
1000
750
500
250
0
0
100
200
300
400
Threads
◆ A768CO
▼ A768NB
▲ A384CO
▶ A384NB

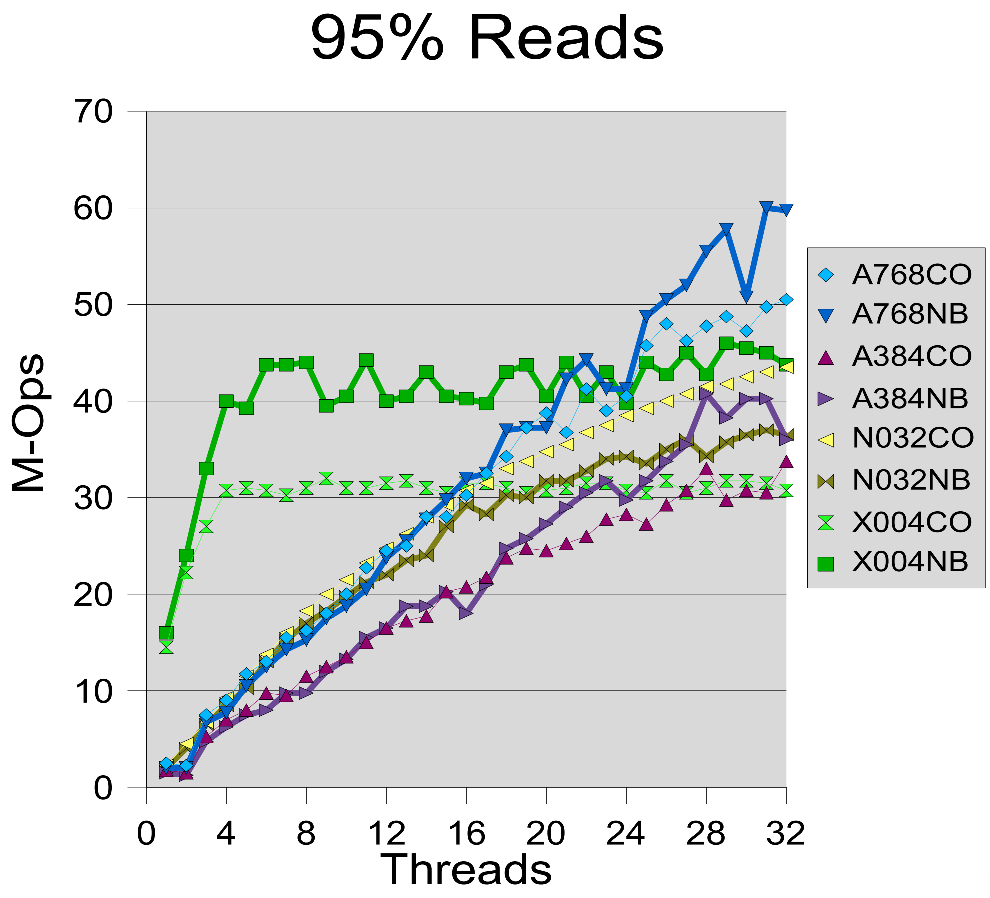
99% Reads


99% Reads
M-Ops
1000
750
500
250
0
0
200
400
600
800
Threads
◆ A768CO
▼ A768NB
▲ A384CO
▶ A384NB

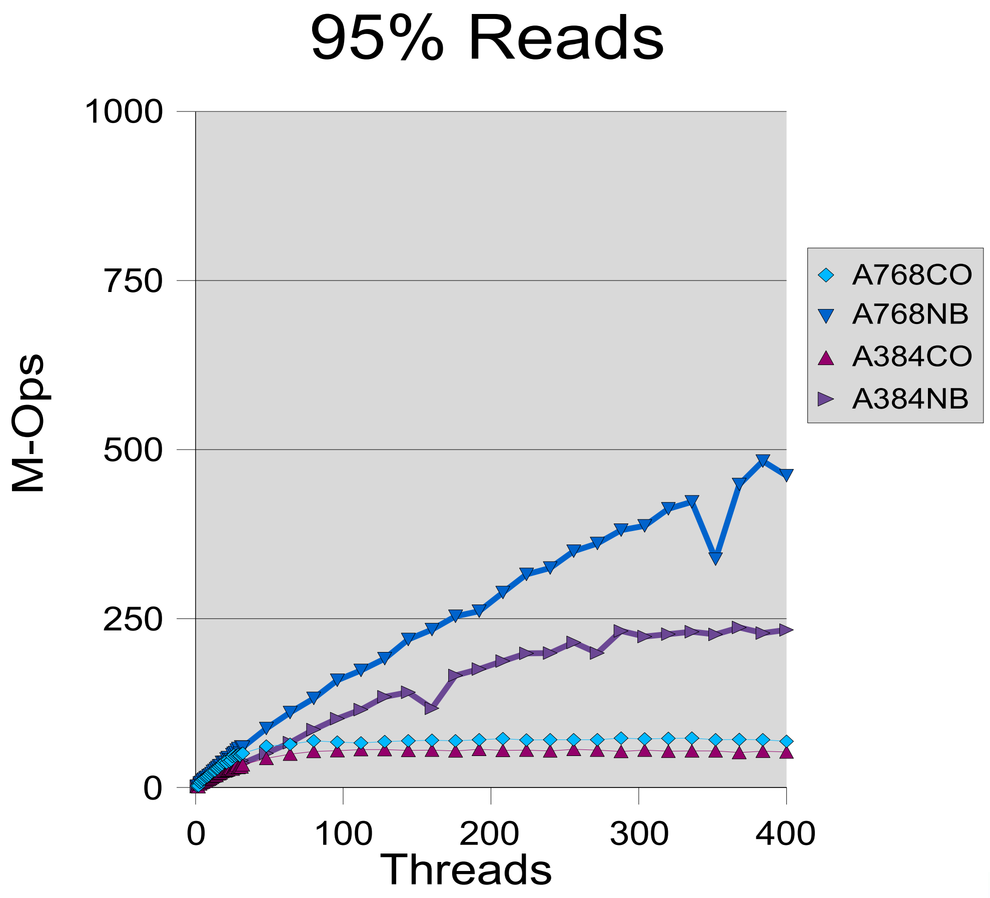
95% Reads


95% Reads
M-Ops
70
60
50
40
30
20
10
0
1
2
3
4
5
6
7
8
Threads
A768CO
A768NB
A384CO
A384NB
N032CO
N032NB
X004CO
X004NB
S002CO
S002NB

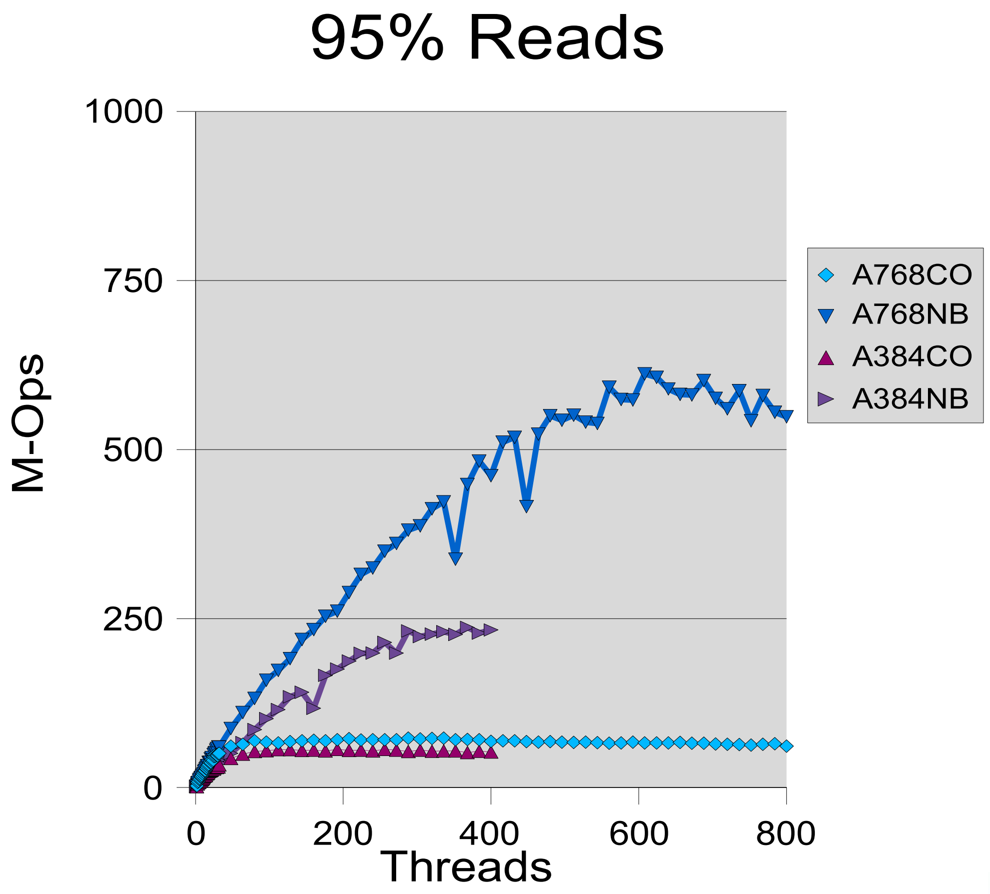
95% Reads


95% Reads
M-Ops
0
10
20
30
40
50
60
70
Threads
0
4
8
12
16
20
24
28
32
A768CO
A768NB
A384CO
A384NB
N032CO
N032NB
X004CO
X004NB

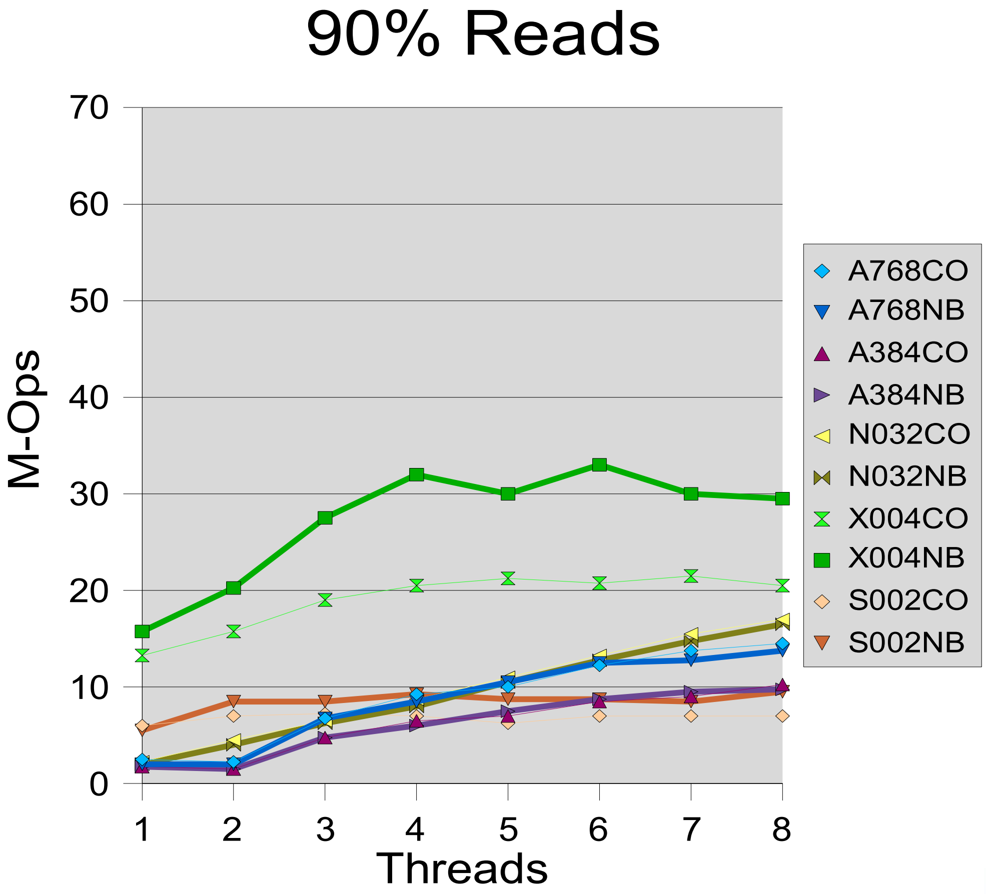
95% Reads


95% Reads
M-Ops
1000
750
500
250
0
0
100
200
300
400
Threads
◆ A768CO
▼ A768NB
▲ A384CO
▶ A384NB

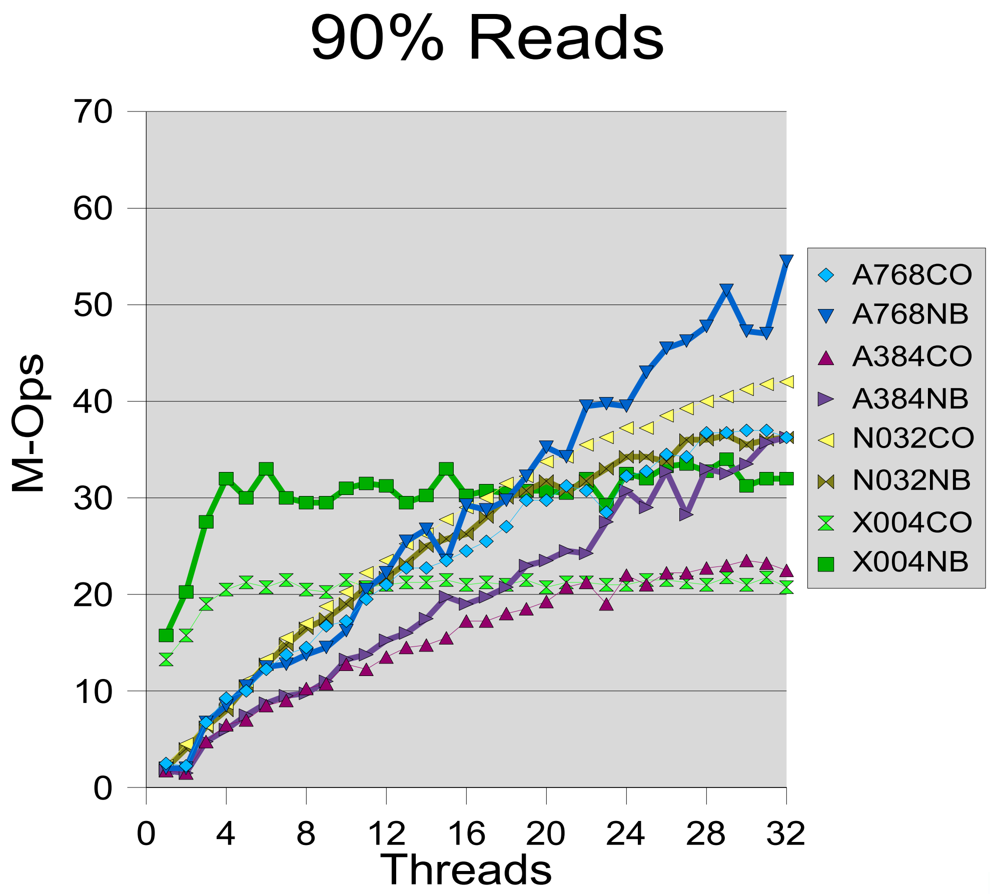
95% Reads


95% Reads
M-Ops
1000
750
500
250
0
0
200
400
600
800
Threads
◆ A768CO
▼ A768NB
▲ A384CO
▶ A384NB

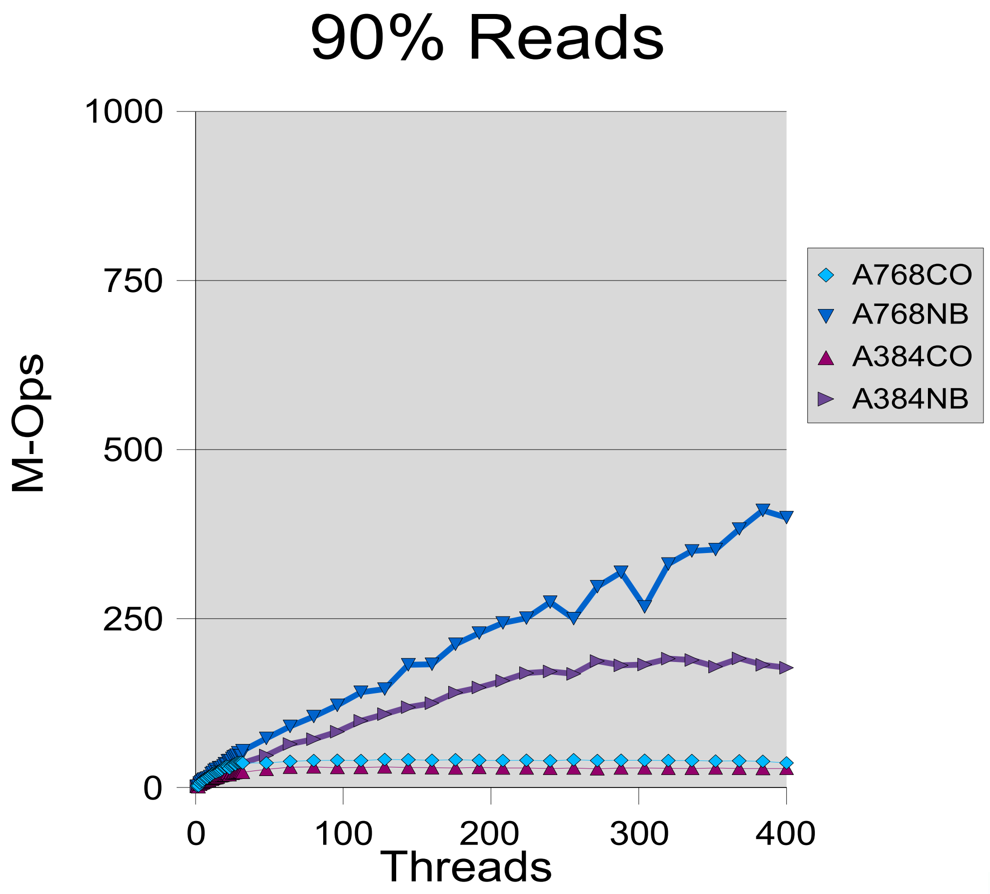
90% Reads


90% Reads
M-Ops
70
60
50
40
30
20
10
0
1
2
3
4
5
6
7
8
Threads
A768CO
A768NB
A384CO
A384NB
N032CO
N032NB
X004CO
X004NB
S002CO
S002NB

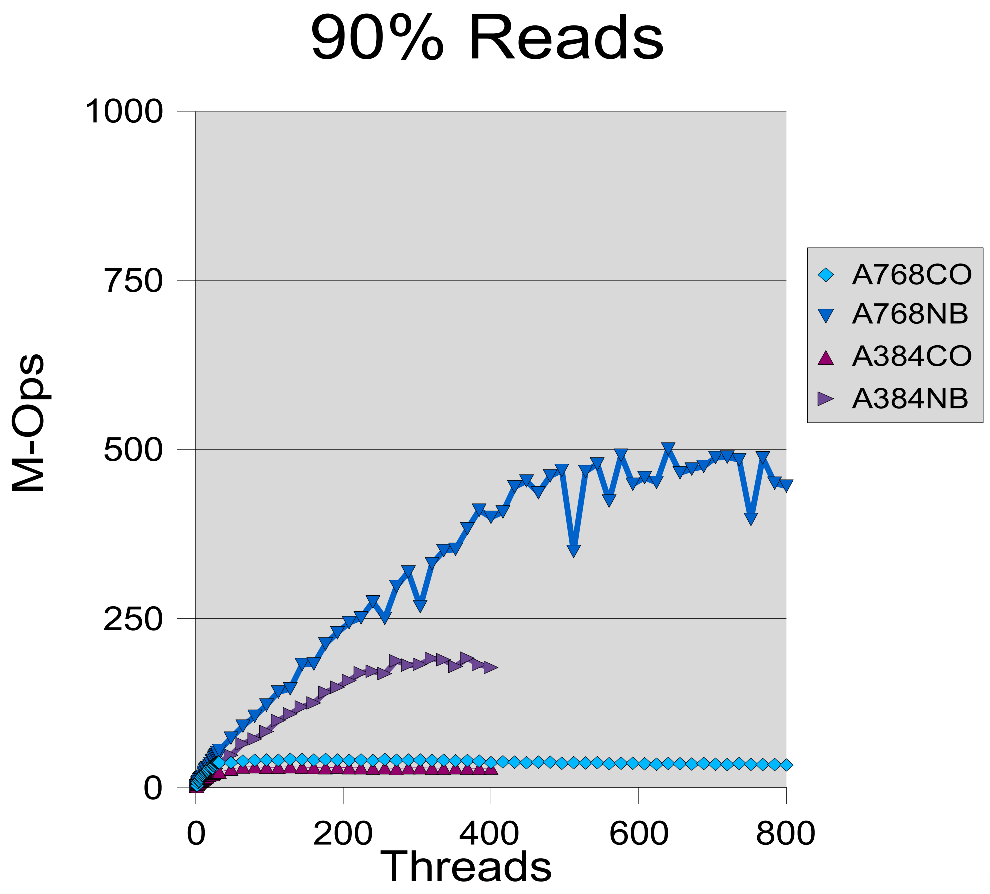
90% Reads


90% Reads
M-Ops
70
60
50
40
30
20
10
0
0
4
8
12
16
20
24
28
32
Threads
A768CO
A768NB
A384CO
A384NB
N032CO
N032NB
X004CO
X004NB

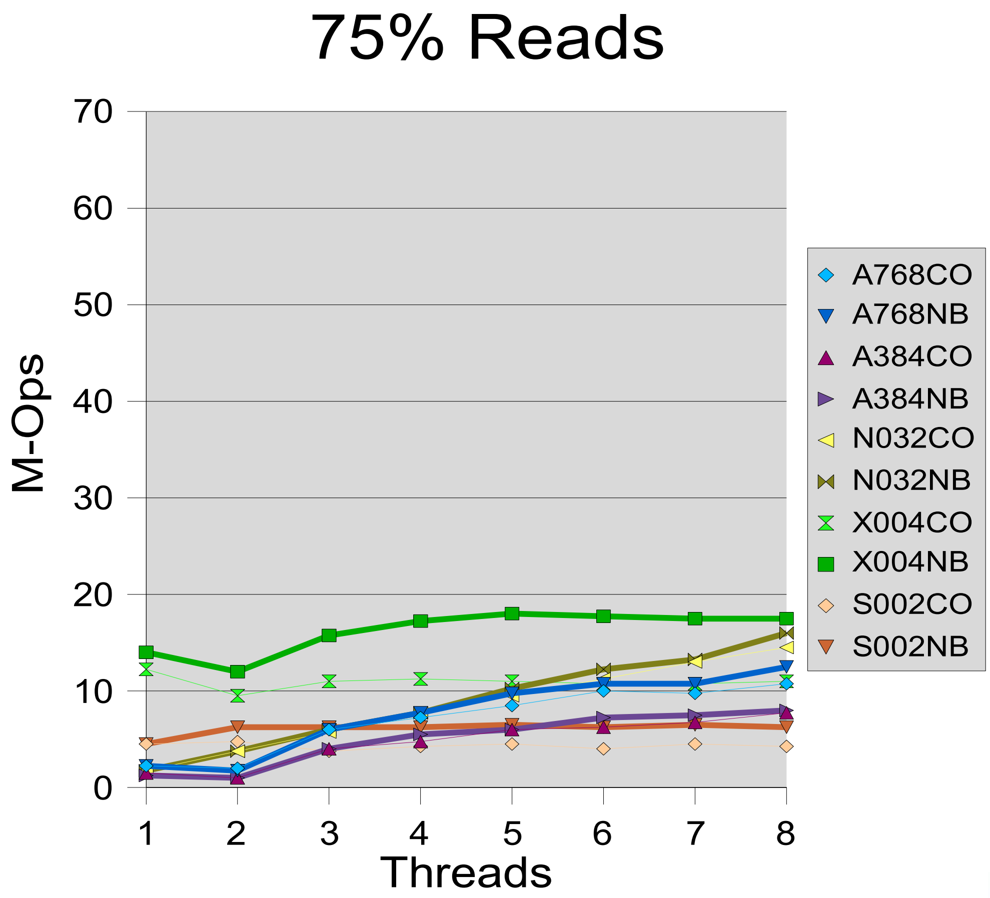
90% Reads


90% Reads
M-Ops
1000
750
500
250
0
0
100
200
300
400
Threads
A768CO
A768NB
A384CO
A384NB

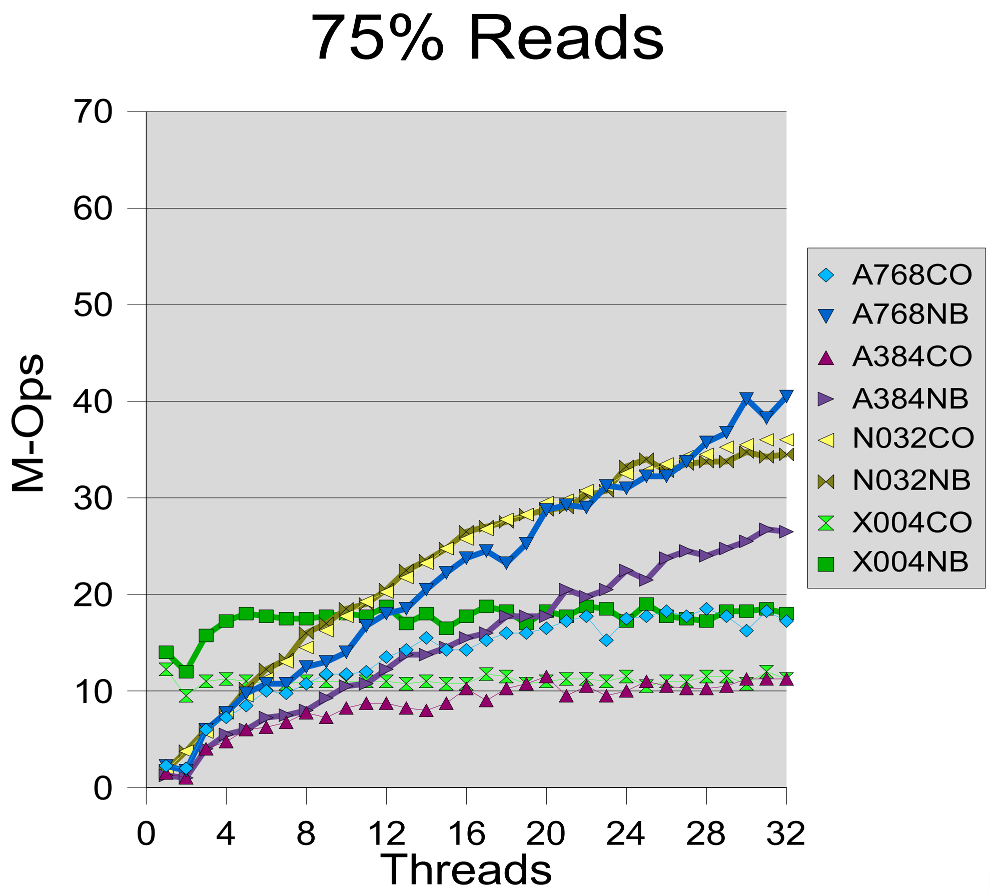
90% Reads


90% Reads
M-Ops
1000
750
500
250
0
0
200
400
600
800
Threads
◆ A768CO
▼ A768NB
▲ A384CO
▶ A384NB

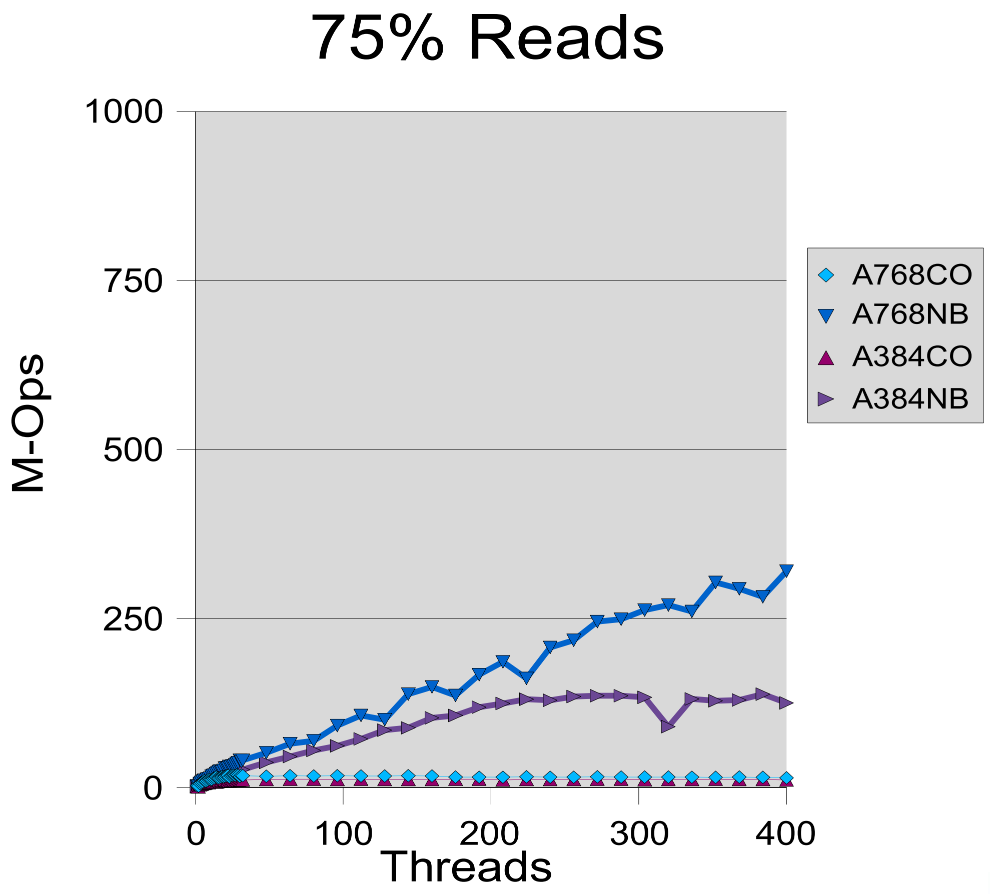
75% Reads


75% Reads
M-Ops
70
60
50
40
30
20
10
0
1
2
3
4
5
6
7
8
Threads
A768CO
A768NB
A384CO
A384NB
N032CO
N032NB
X004CO
X004NB
S002CO
S002NB

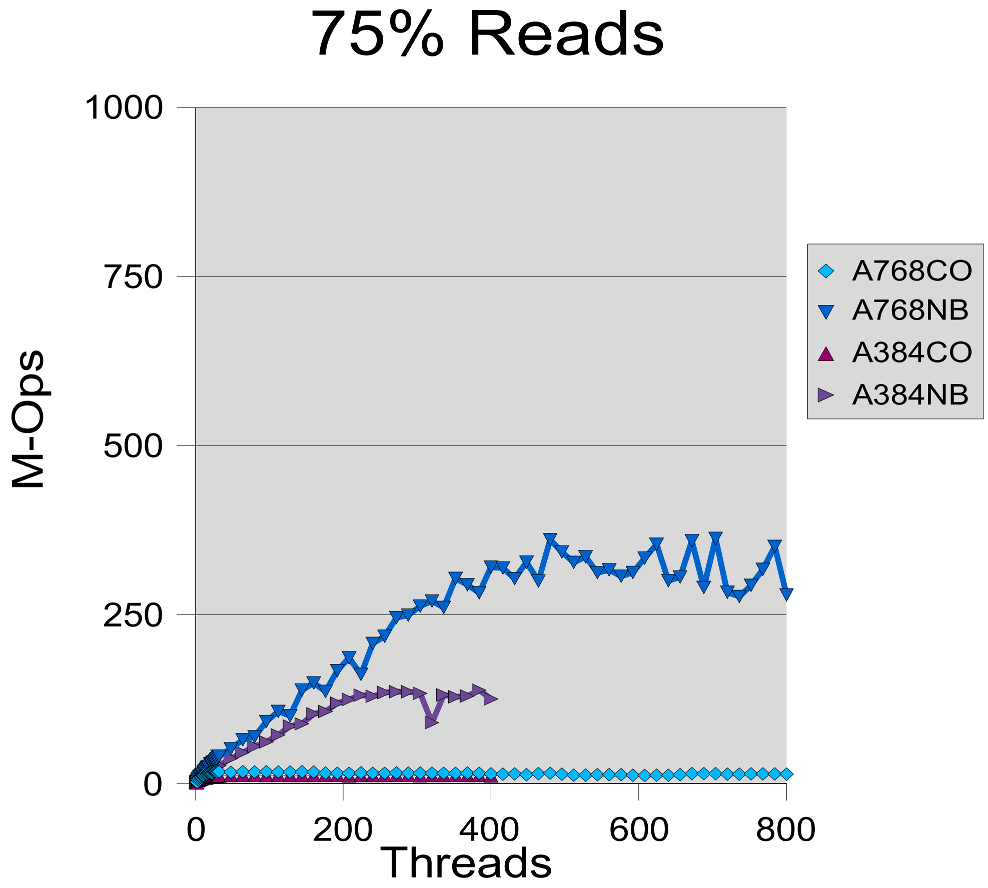
75% Reads


75% Reads
M-Ops
70
60
50
40
30
20
10
0
0
4
8
12
16
20
24
28
32
Threads
A768CO
A768NB
A384CO
A384NB
N032CO
N032NB
X004CO
X004NB

75% Reads


75% Reads
M-Ops
1000
750
500
250
0
0
100
200
300
400
Threads
◆ A768CO
▼ A768NB
▲ A384CO
▶ A384NB

75% Reads


75% Reads
1000
750
500
250
0
M-Ops
0
200
400
600
800
Threads
◆ A768CO
▼ A768NB
▲ A384CO
▶ A384NB

Summary

- A faster lock-free *wait-free* HashTable
- Faster for more CPUs
- Much faster for higher table modification rate
- State-Based Reasoning:
  - No ordering, no JMM, no fencing
- Seems applicable to other data structures as well
  - Have a concurrent j.u.Vector in the works

http://www.azulsystems.com/events/stanford_2007/index.htm

AZUL SYSTEMS

UNSHACKLE THE POWER OF JAVA

NETWORK ATTACHED PROCESSING FROM AZUL SYSTEMS

Thank you.

cliffc@acm.org

"Uninteresting" Resizing Control

- Each old slot copied exactly once
- Update with CAS to indicate copy
- Still need efficient worklist control
  - Chunk K/V pairs to copy
  - CAS out work chunks
- Wait-Free: no CAS loops
  - Try CAS a few times, then quit helping
  - And proceed with other work
  - Since CAS failed, other threads are copying

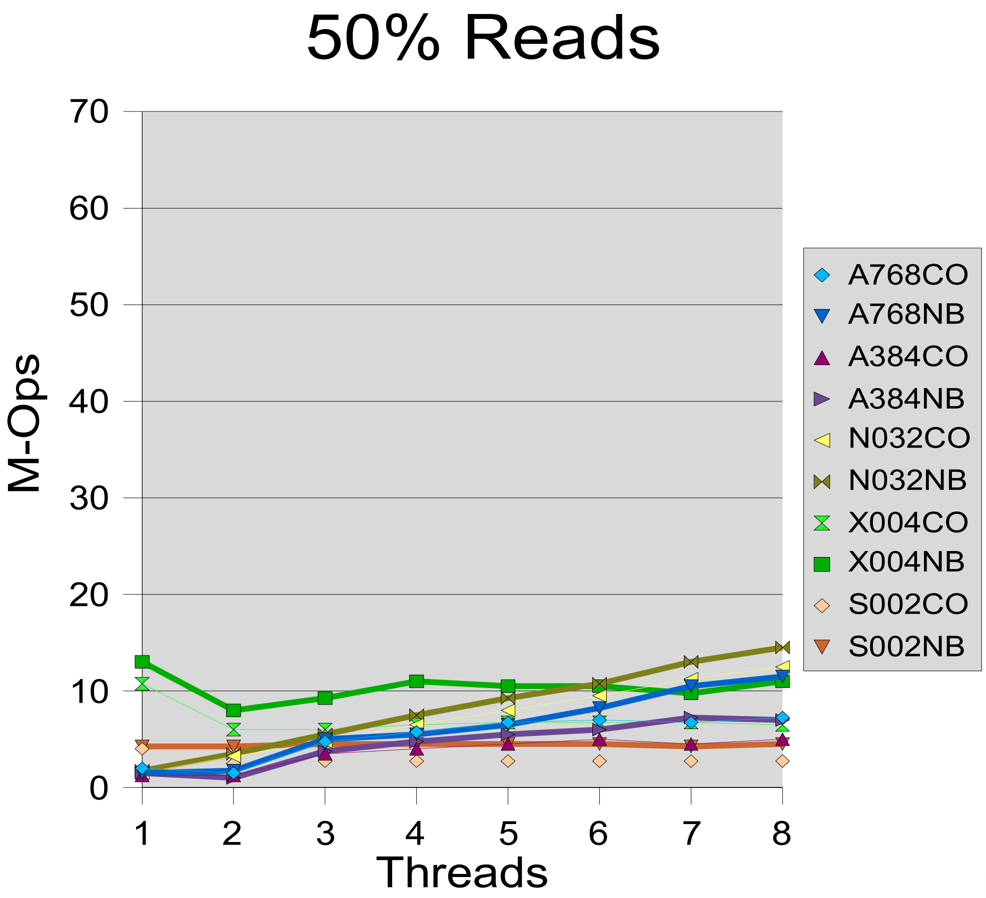
Wait-Free

- Requires “no spurious failure” CAS
- No CAS spin-loops
  - Lest you wait forever for success
- Try CAS once
  - If fails – must be contention
  - i.e., Another racing writer is writing
  - Allow other writer to win
- “As If” this write succeeded but was immediately overwritten by another racing writer

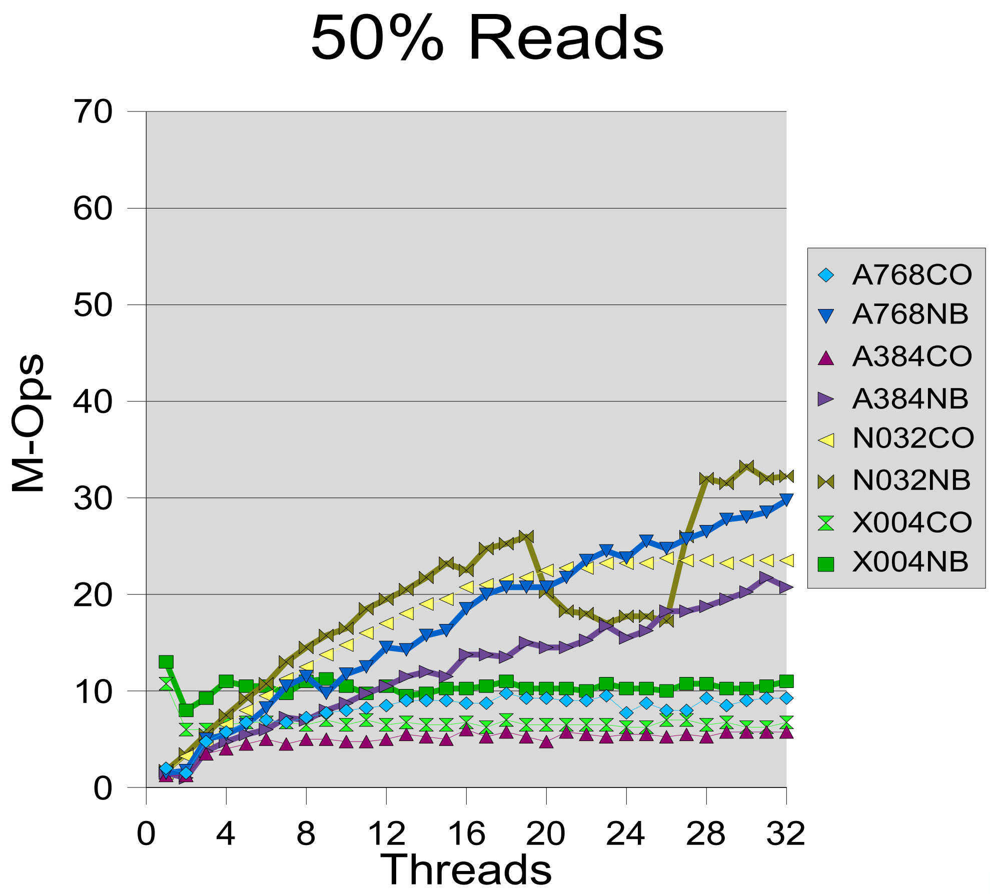
Obstruction-Free

- Obstruction-Free: no thread stalled forever
- Resize may stall:
  - Copy in-progress slows down table by O(1)
  - Throbbing in old table can prevent copy
  - But only for put's started before resize started
  - Limited by #threads doing a “late put()”

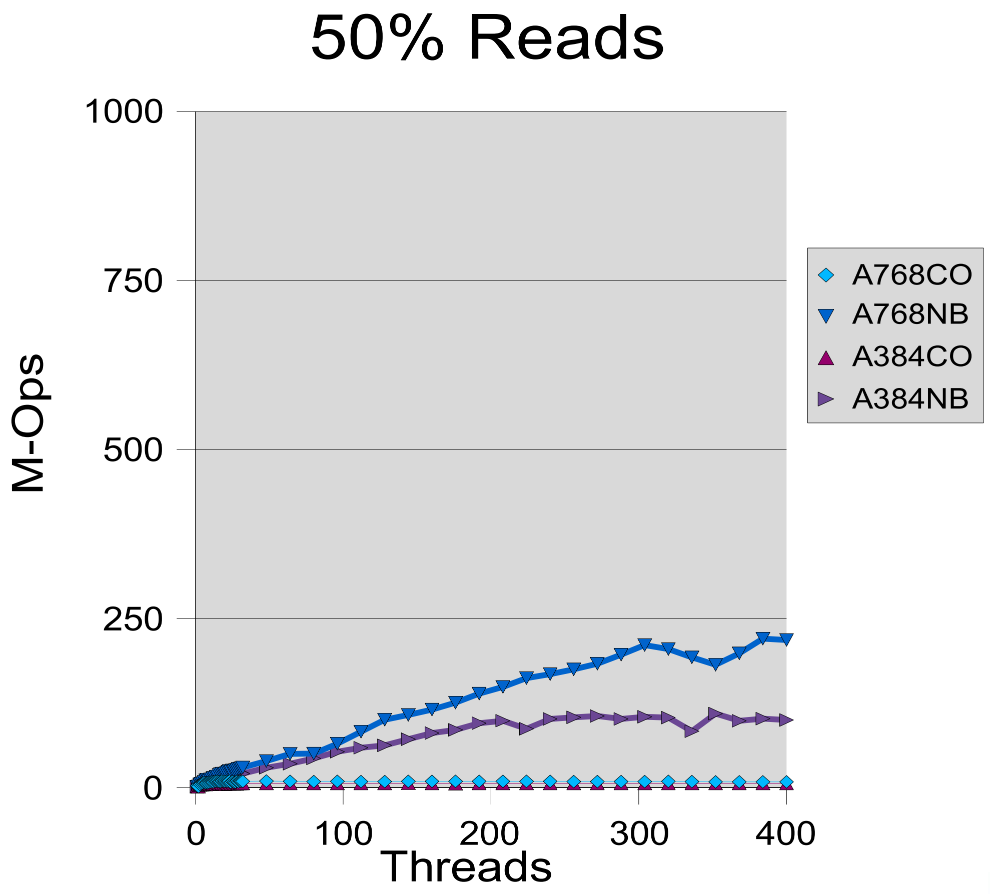
50% Reads


50% Reads
M-Ops
70
60
50
40
30
20
10
0
1
2
3
4
5
6
7
8
Threads
A768CO
A768NB
A384CO
A384NB
N032CO
N032NB
X004CO
X004NB
S002CO
S002NB

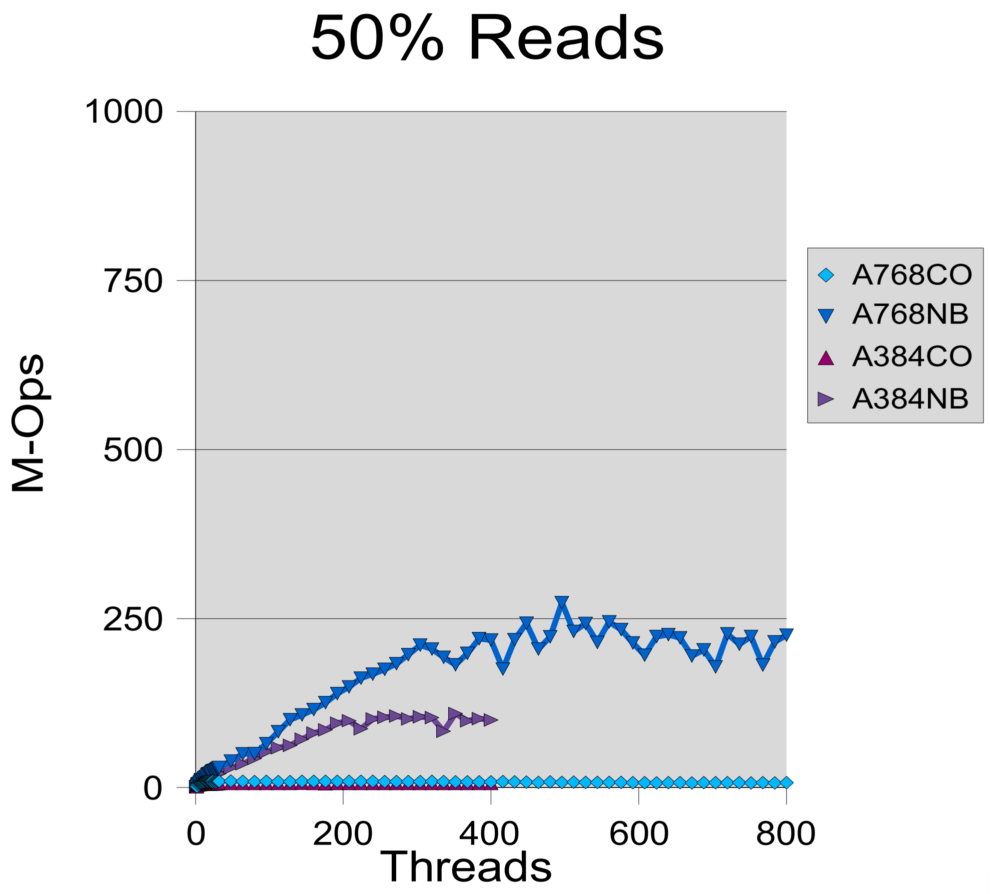
50% Reads


50% Reads
M-Ops
70
60
50
40
30
20
10
0
0
4
8
12
16
20
24
28
32
Threads
A768CO
A768NB
A384CO
A384NB
N032CO
N032NB
X004CO
X004NB

50% Reads


50% Reads
M-Ops
1000
750
500
250
0
0
100
200
300
400
Threads
◆ A768CO
▼ A768NB
▲ A384CO
▶ A384NB

50% Reads


50% Reads
M-Ops
1000
750
500
250
0
0
200
400
600
800
Threads
◆ A768CO
▼ A768NB
▲ A384CO
▶ A384NB