



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
 - 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/put
- Auto-Resize

“Uninteresting” Details

- Example get work:

```
idx = hash = key.hashCode();  
while( true ) {           // reprobng 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  
}
```

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

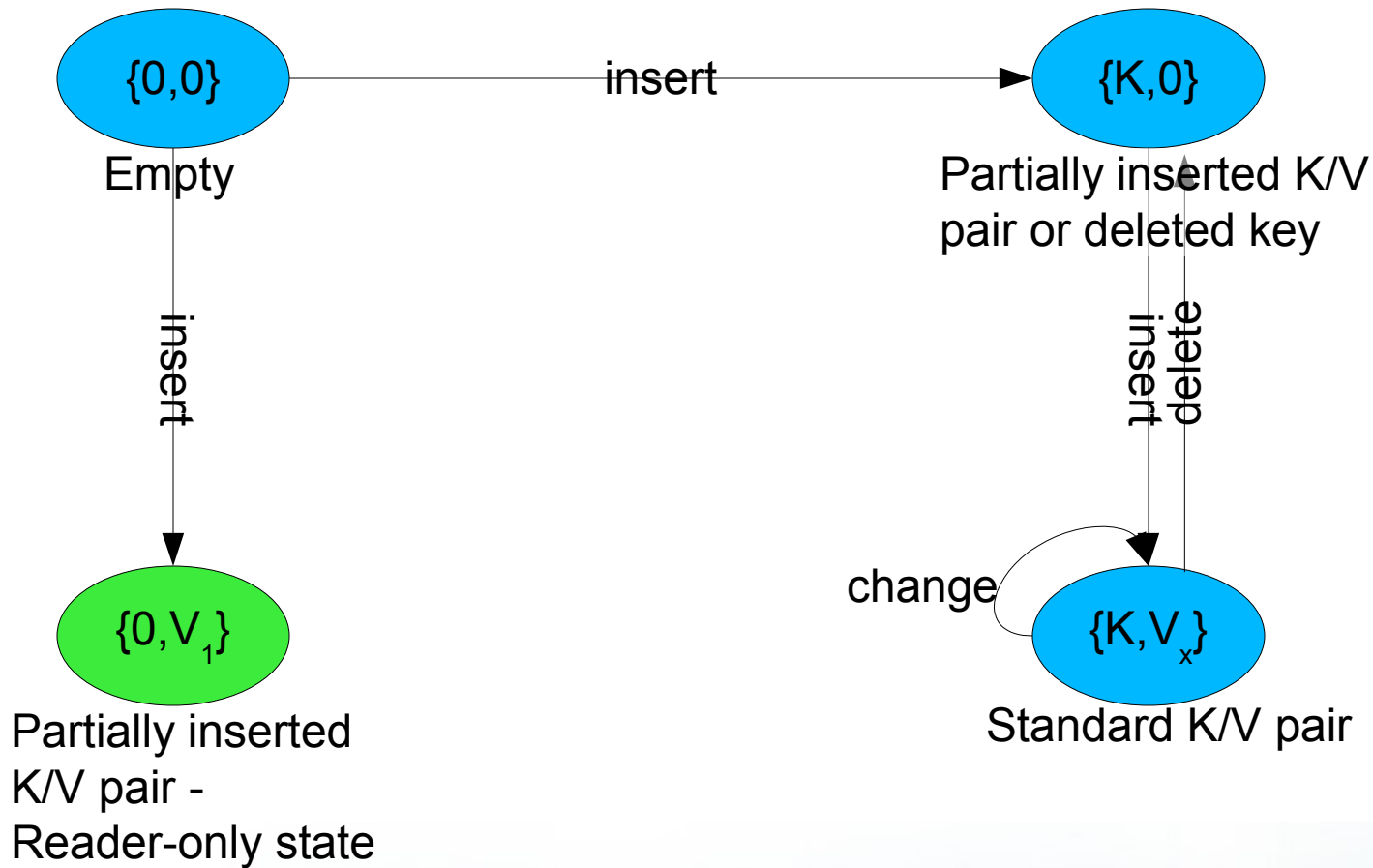
State-Based Reasoning

- 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:
 - 0 – empty
 - K – Some Key; can never change again
- A Value slot is:
 - 0 – empty
 - V_1 – Some Value
 - V_2 – Some other Value
- A state is a {Key, Value} pair

State Machine



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 $\{0, V\}$ state meaningful but uninteresting
 - Means reader got a null 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 synch
 - 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 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

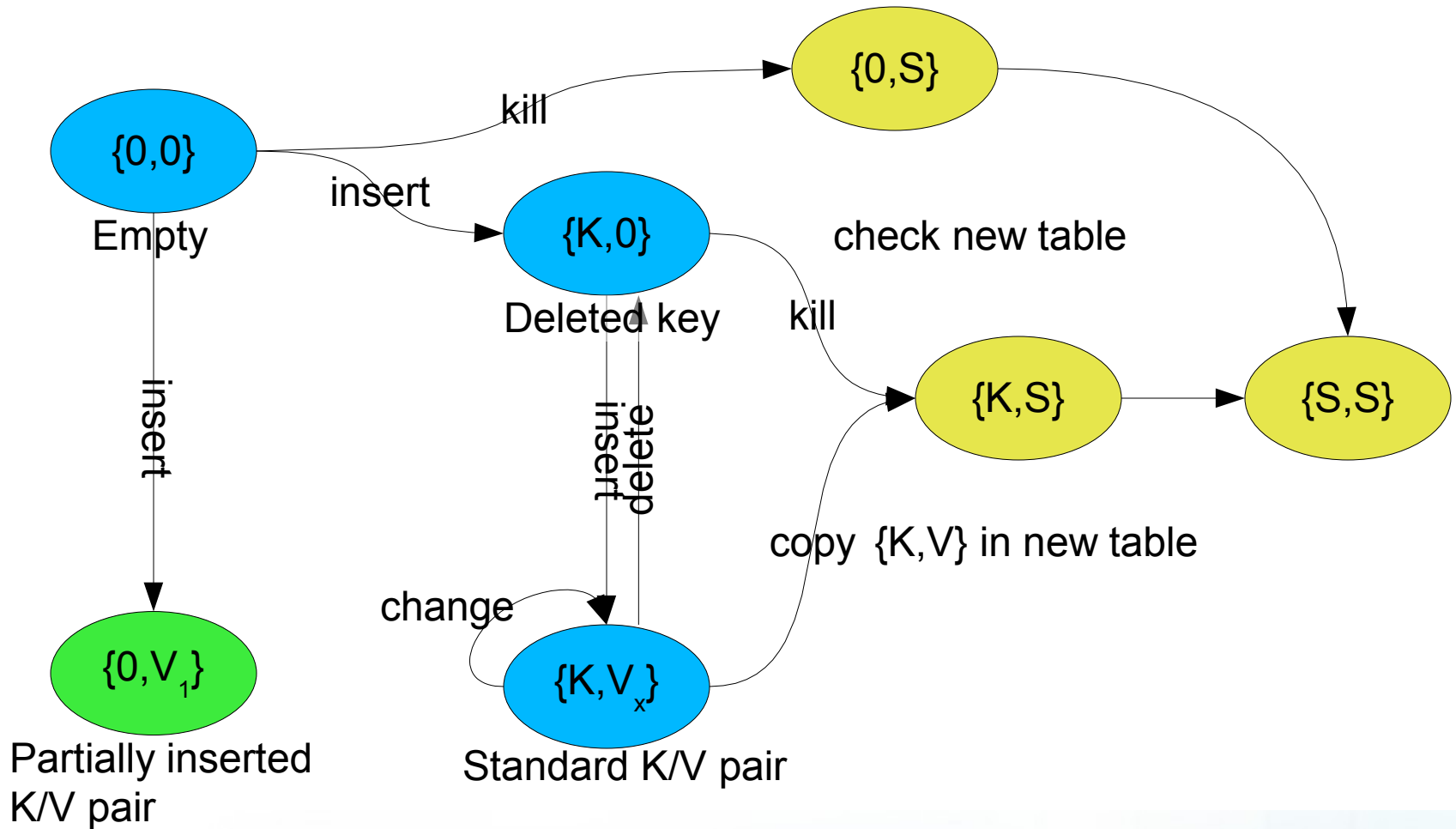
Resizing

- 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:
 - 0 – empty
 - K – Some unchanging Key
 - S – Sentinel, not any valid Key
- A Value slot is:
 - 0 – empty
 - V_x – Some Values
 - S – Sentinel, not any valid Value

State Machine



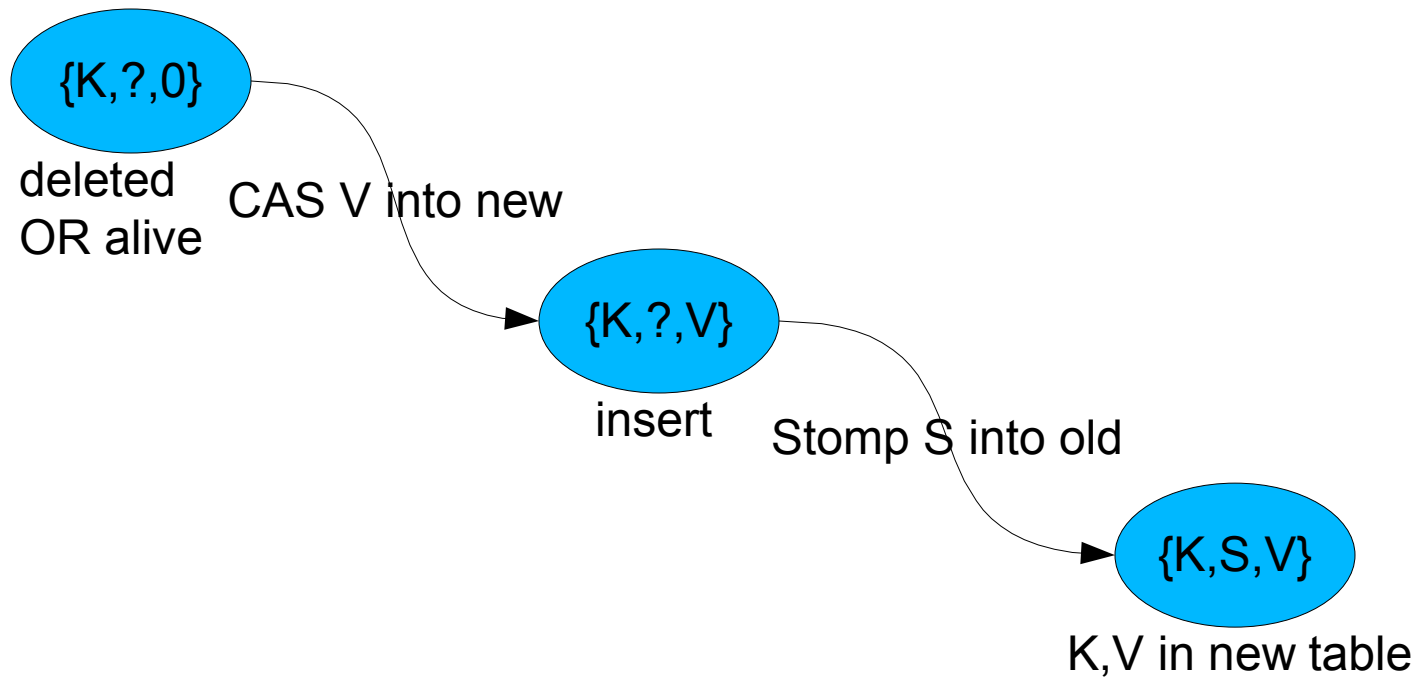
Resizing

- Live K/V-copy is independent of reader/writer
- Many heuristics to choose from:
 - All touching threads, only writers, unrelated thread(s), etc
- Get works on the old table
 - Unless see a sentinel
- Put/mod *must* use new table
- Must check for new table every time
 - Late writes to old table 'happens before' resize

Resizing – PUT 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
- State Machine may help you visualize...
- New States: {Key,OldVal,NewVal}

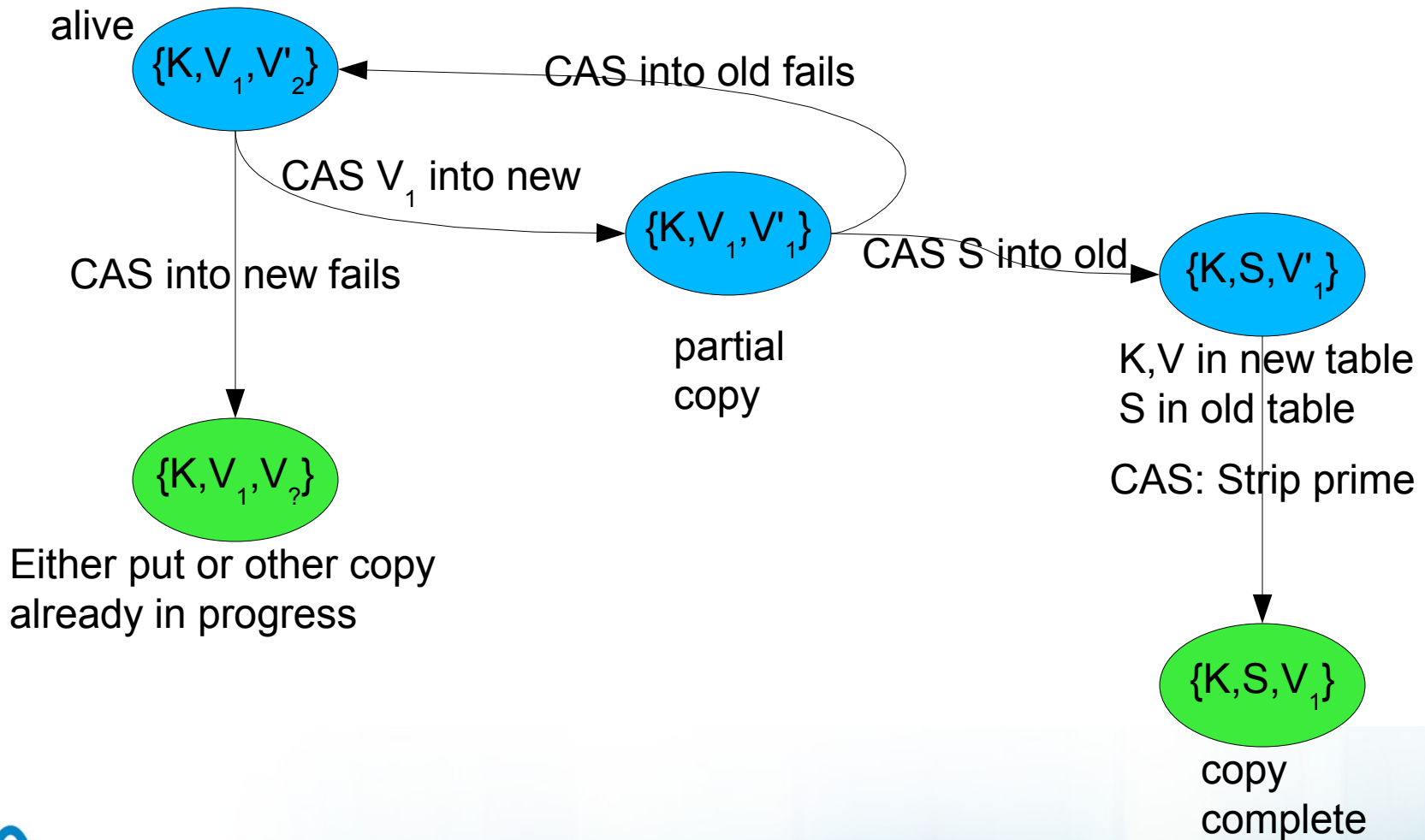
State Machine: PUT while copy



Resizing – Normal Copy

- 'Get' thread or helper thread
- Must make sure to copy late-arriving write
- Attempt to copy atomically
 - May fail & copy does not make progress
 - But old,new tables not damaged
- Read Old, Copy into New, CAS Old to 'S'
- Prime'd values copied from old, always before Put
- State Machine again...

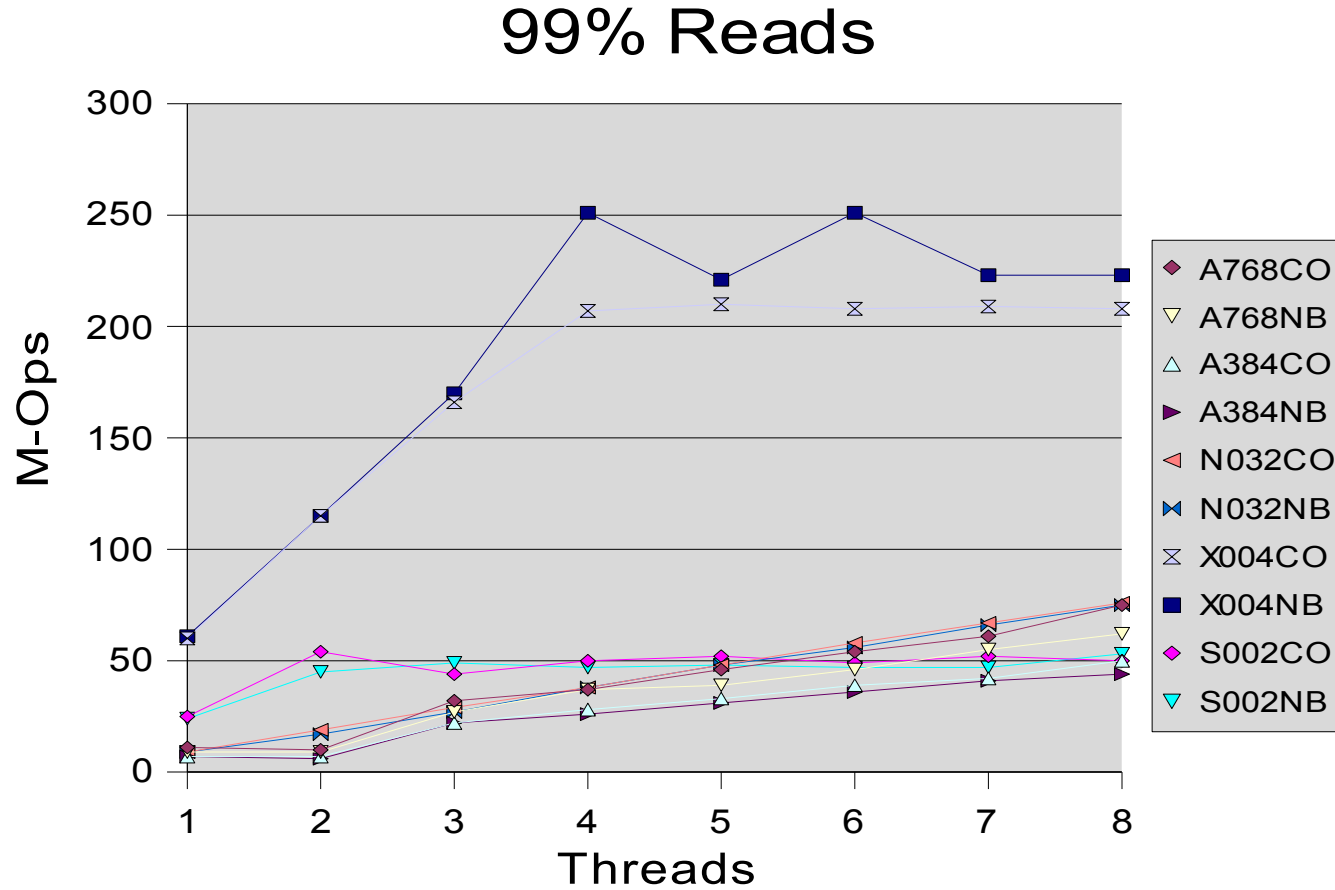
State Machine: Copy One Pair



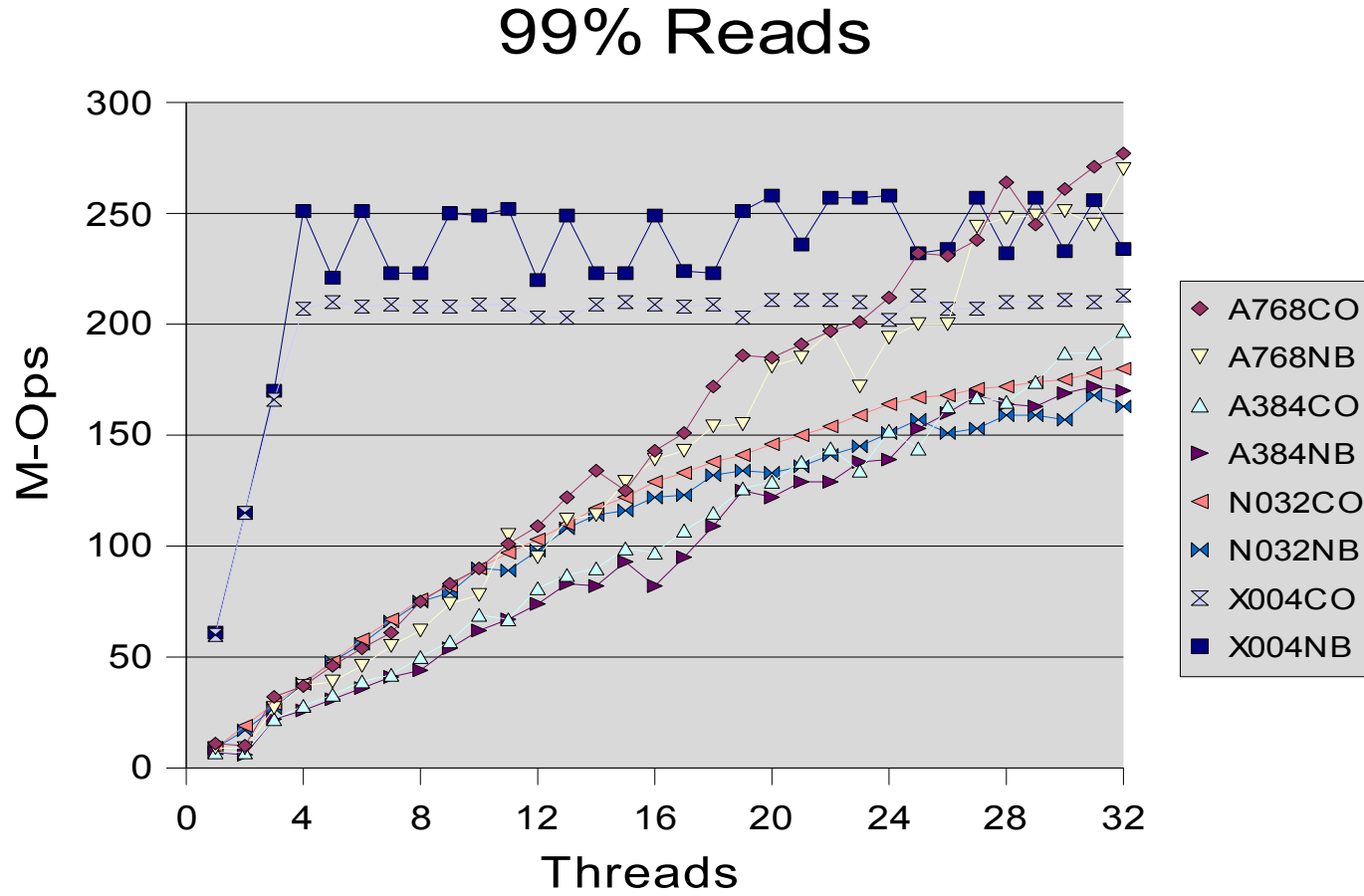
Agenda

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

99% Reads

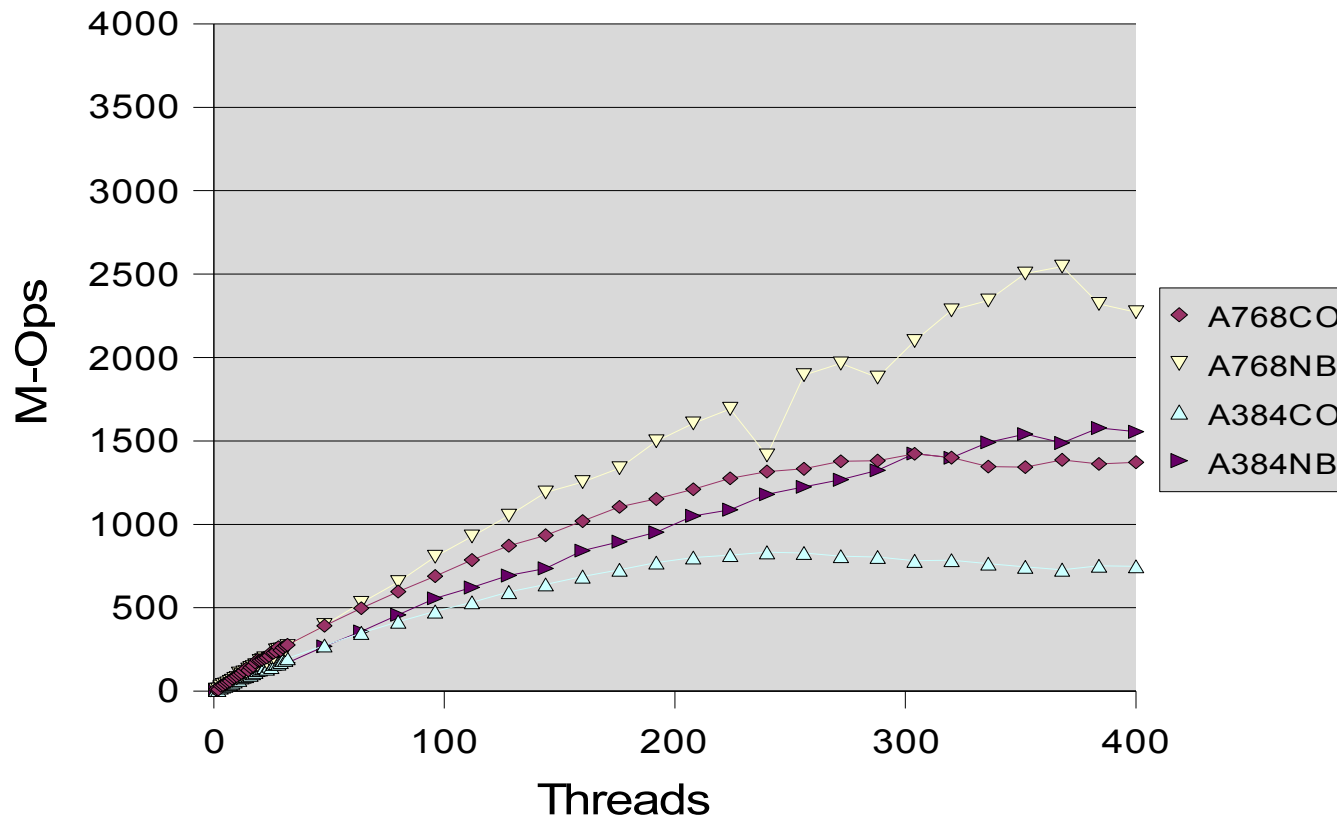


99% Reads

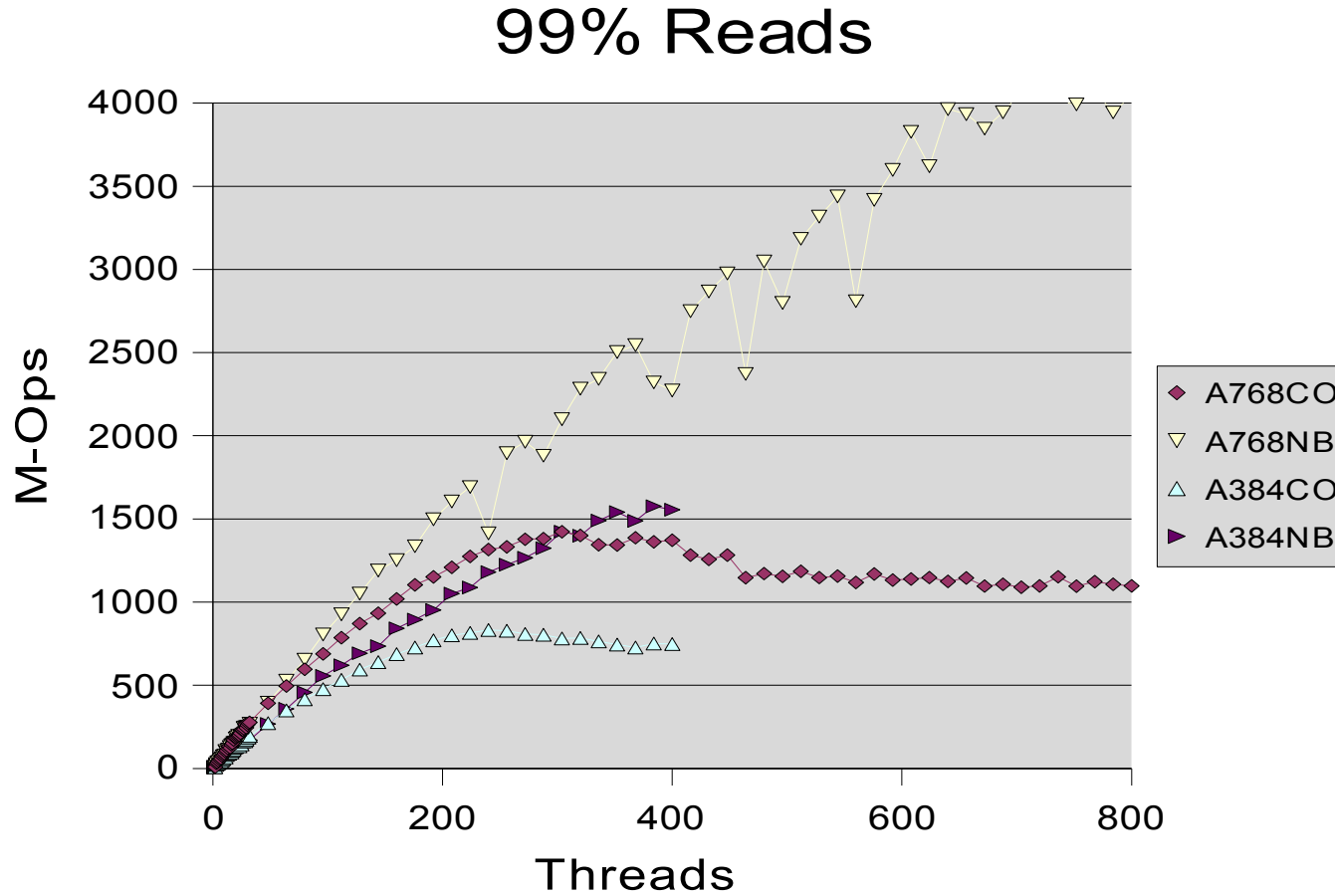


99% Reads

99% Reads

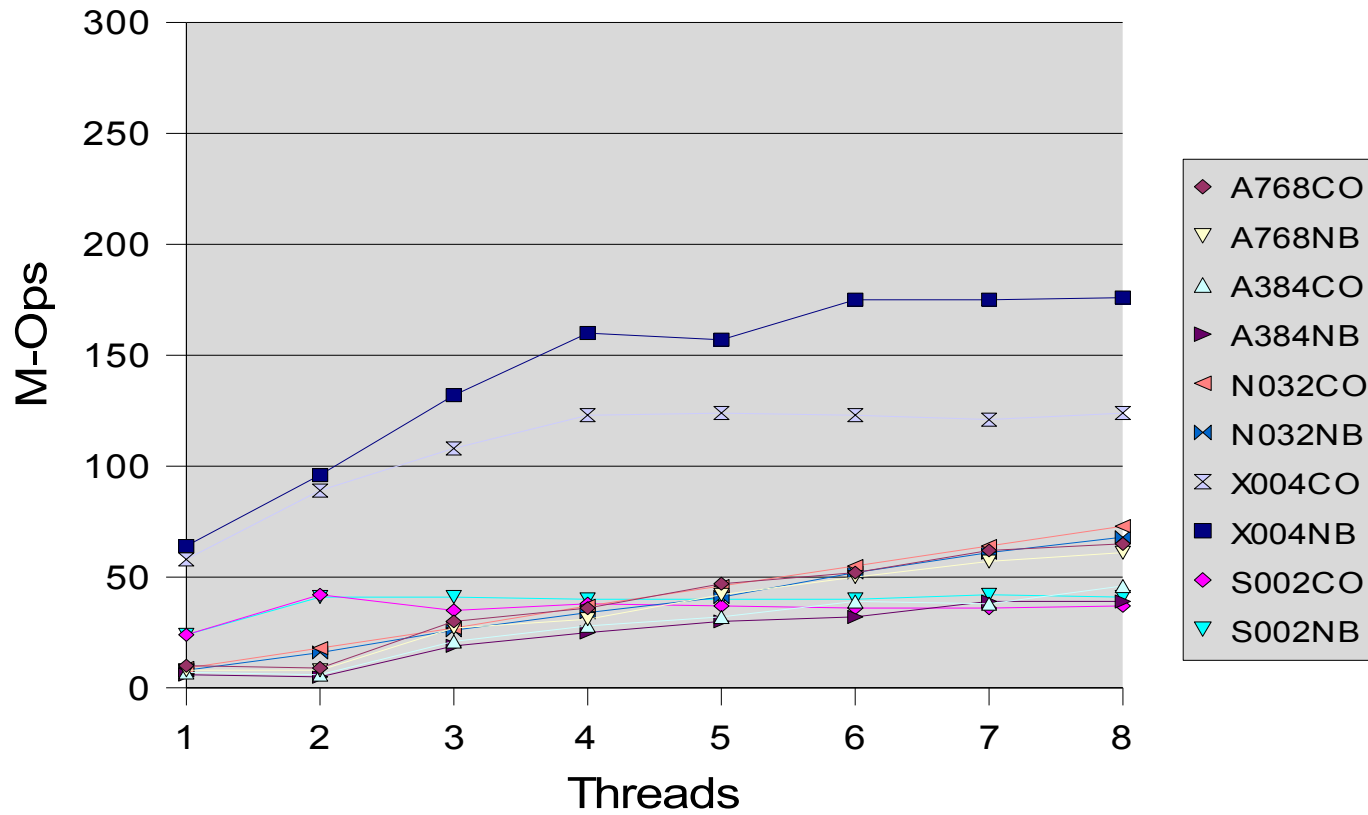


99% Reads



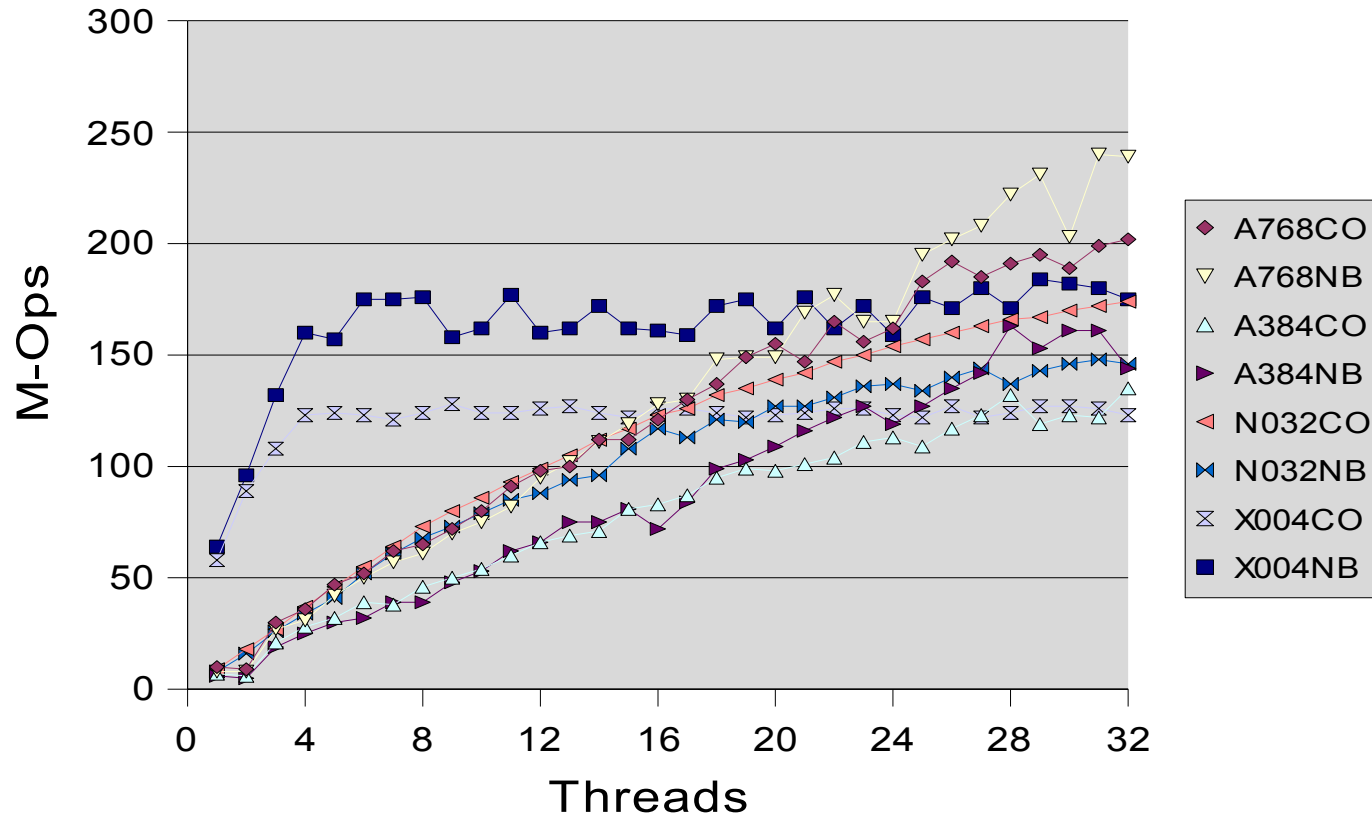
95% Reads

95% Reads



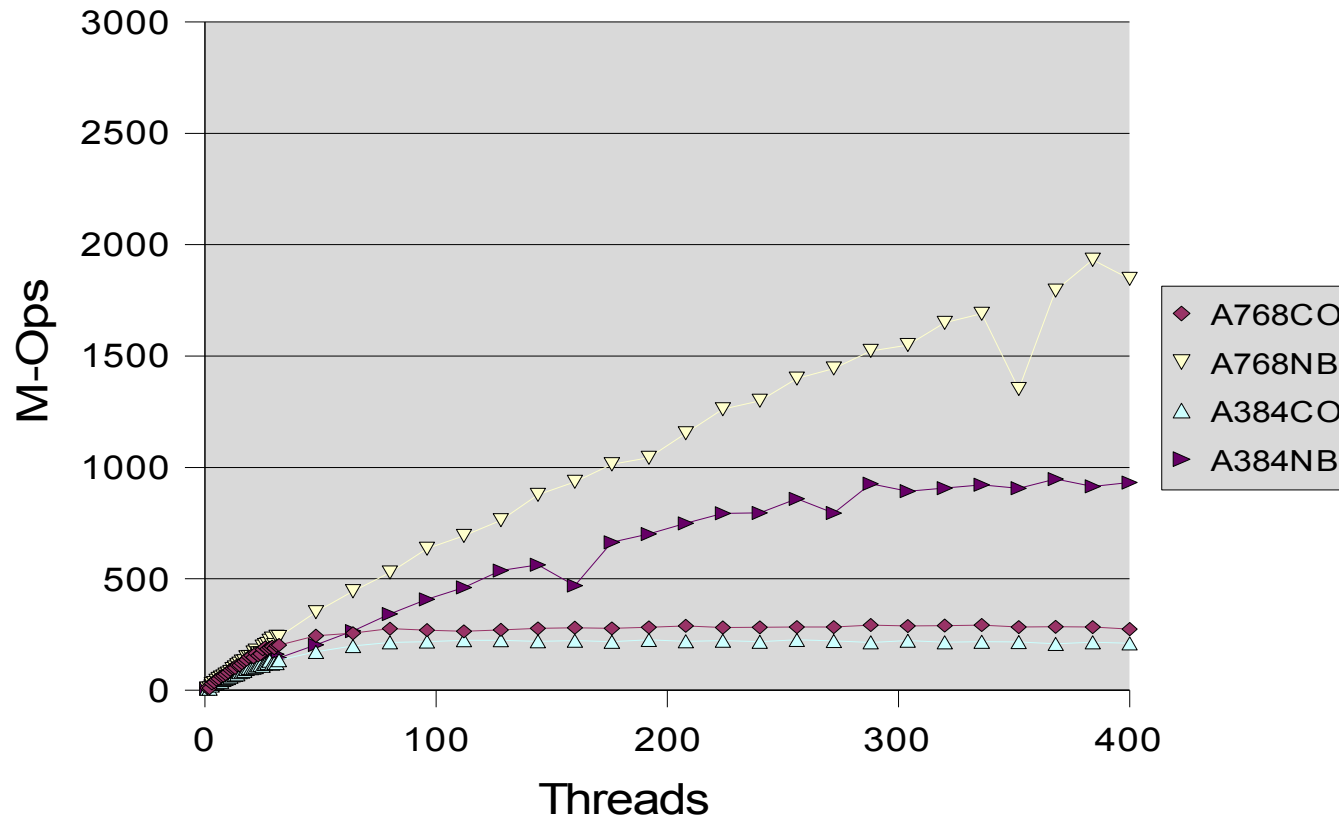
95% Reads

95% Reads

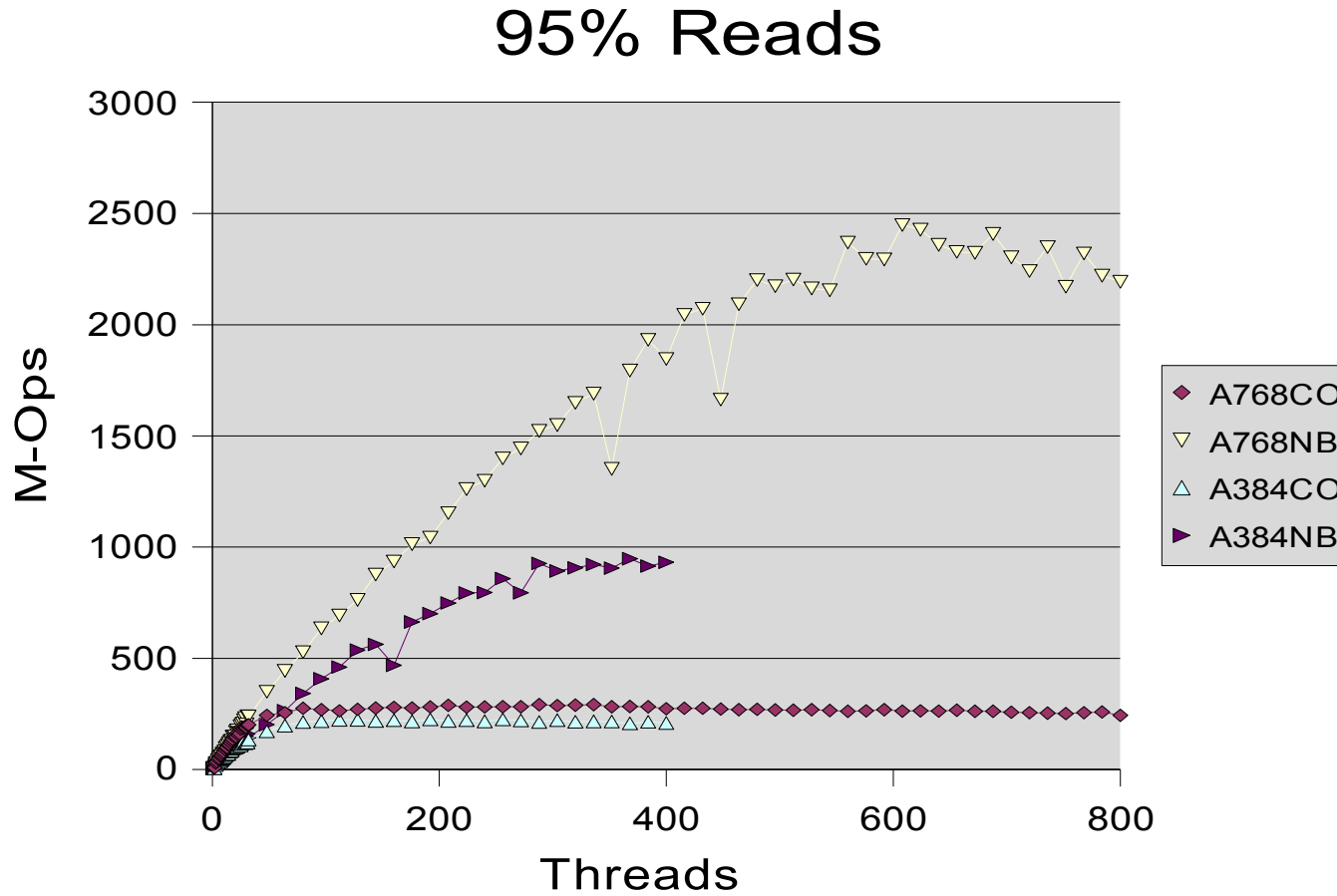


95% Reads

95% Reads

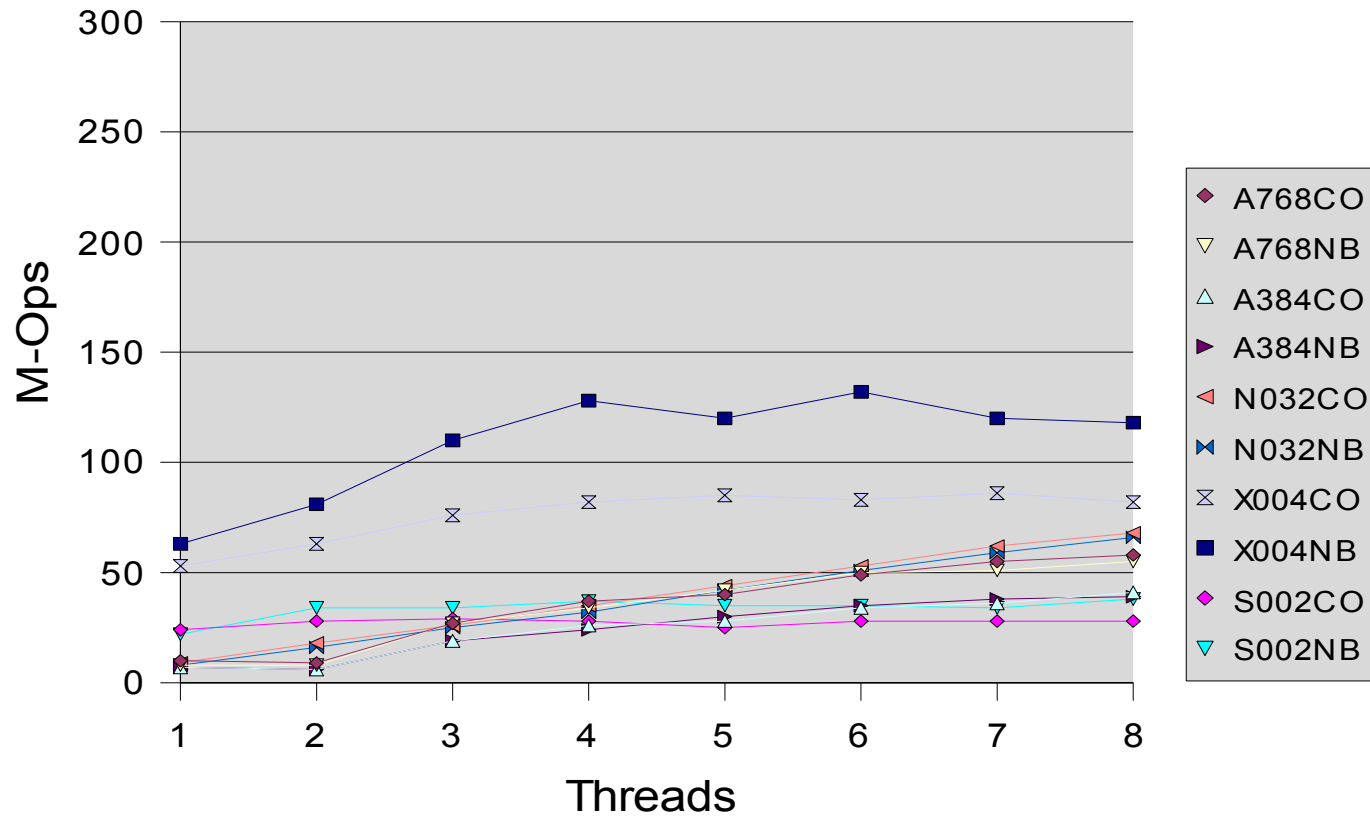


95% Reads



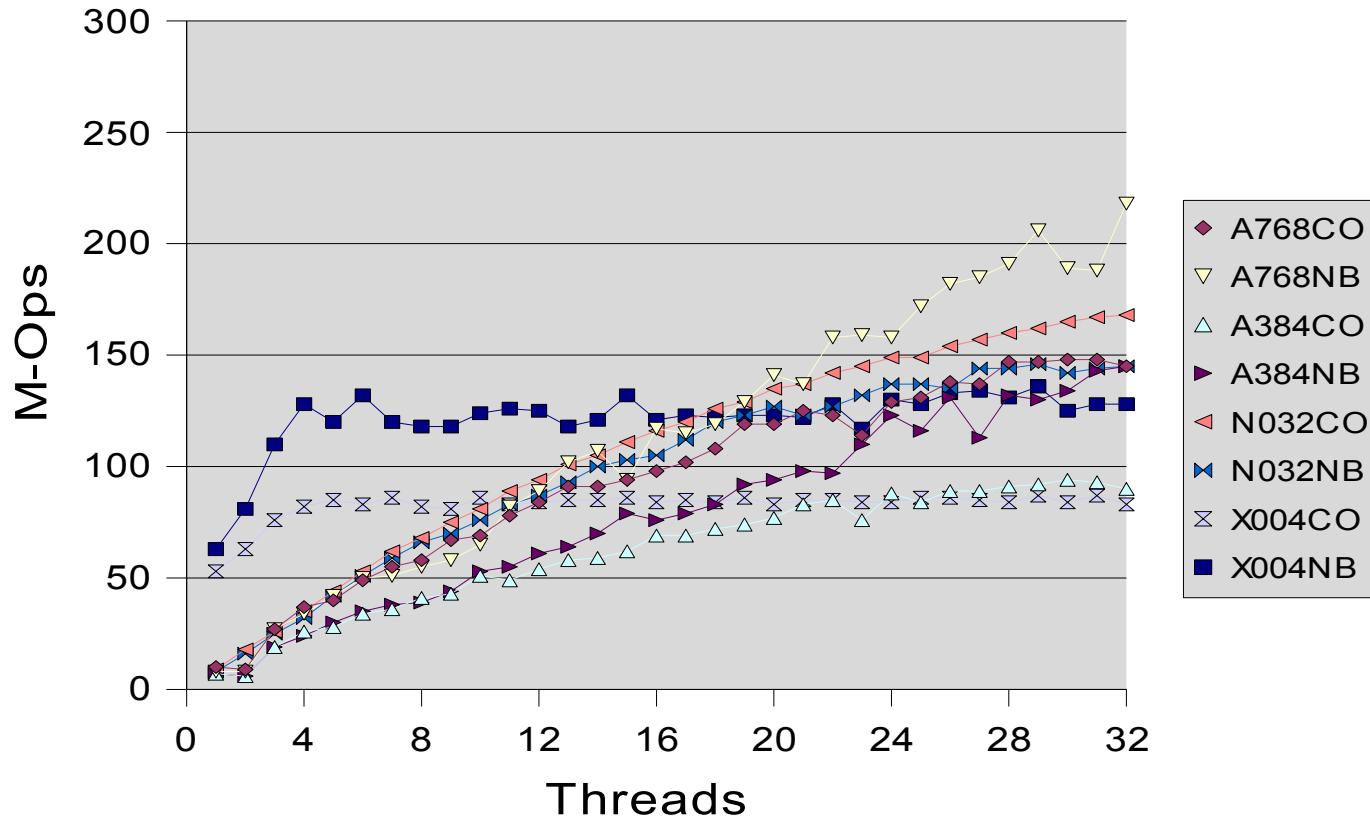
90% Reads

90% Reads



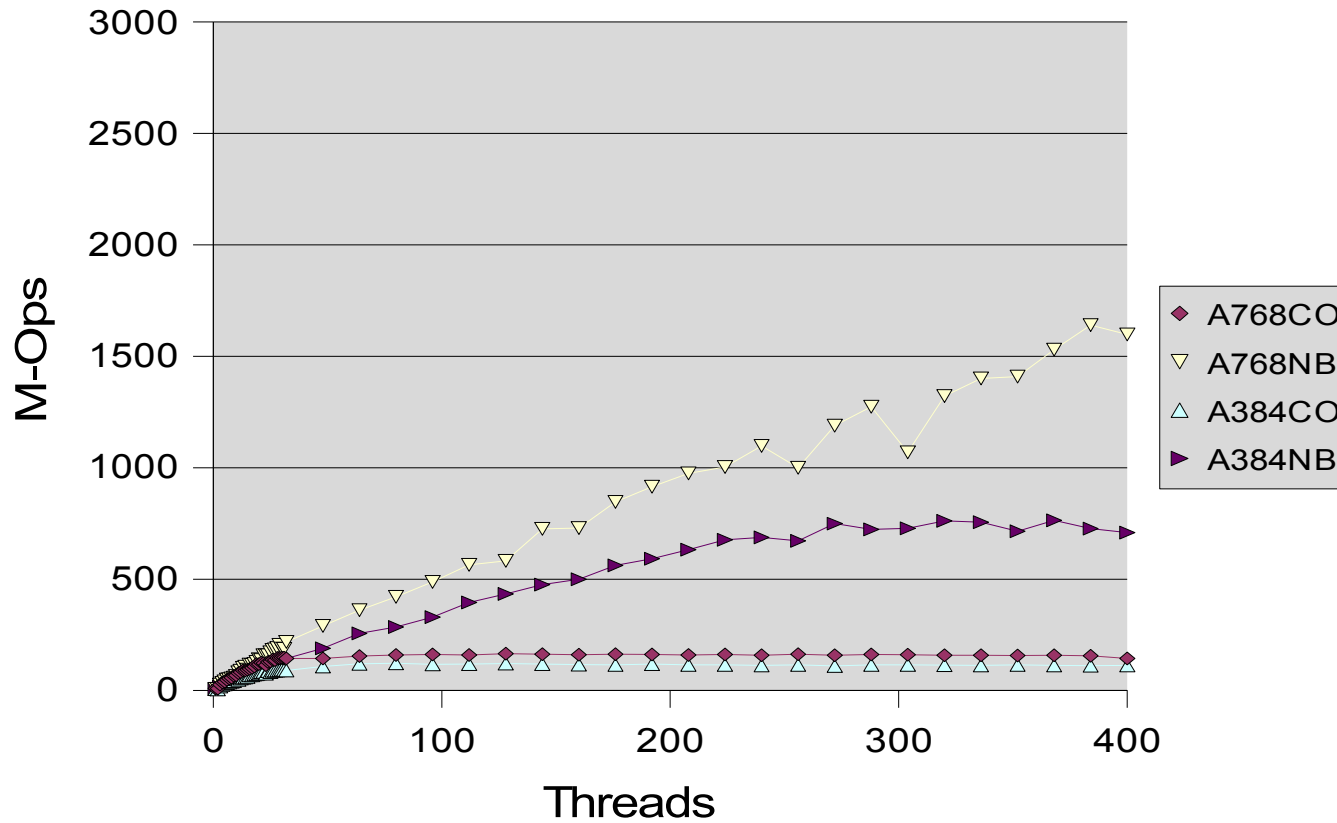
90% Reads

90% Reads



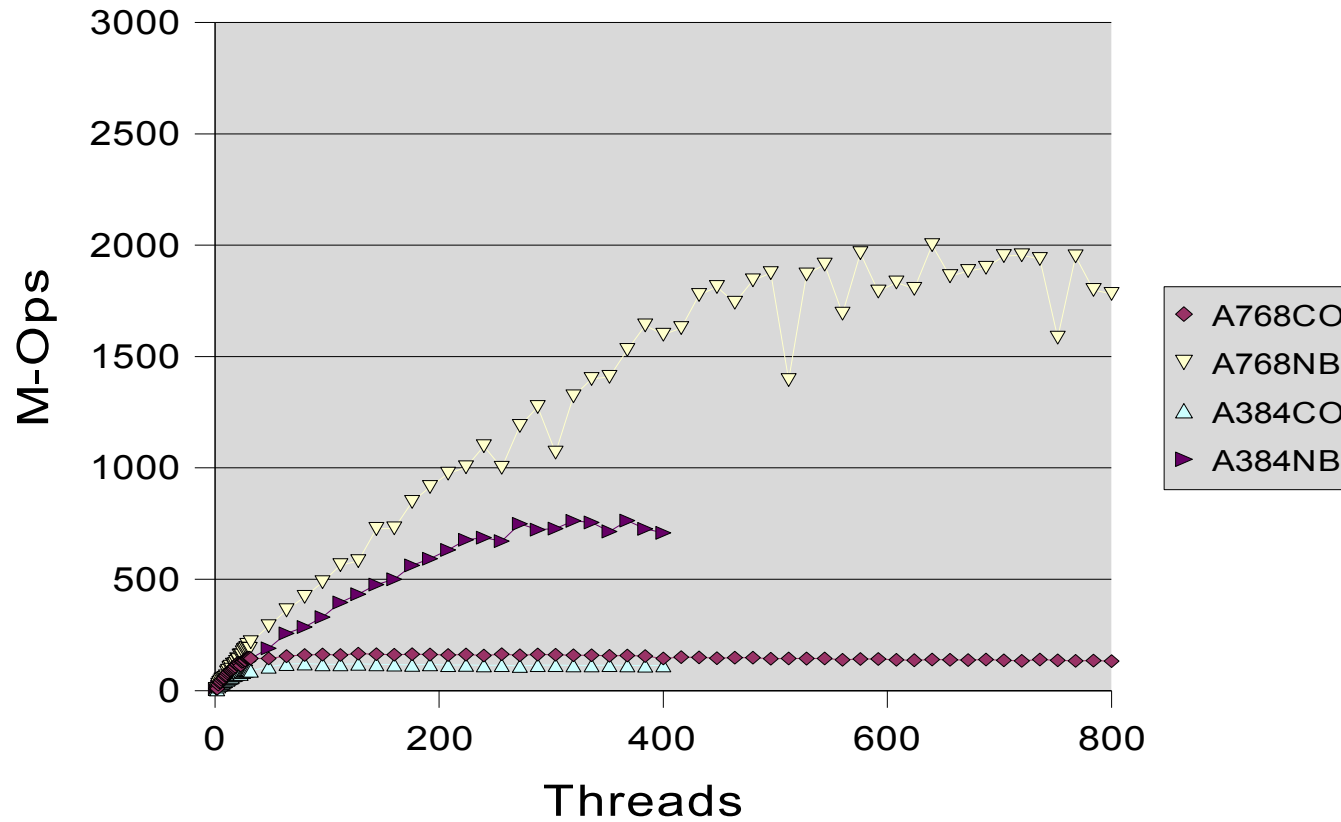
90% Reads

90% Reads



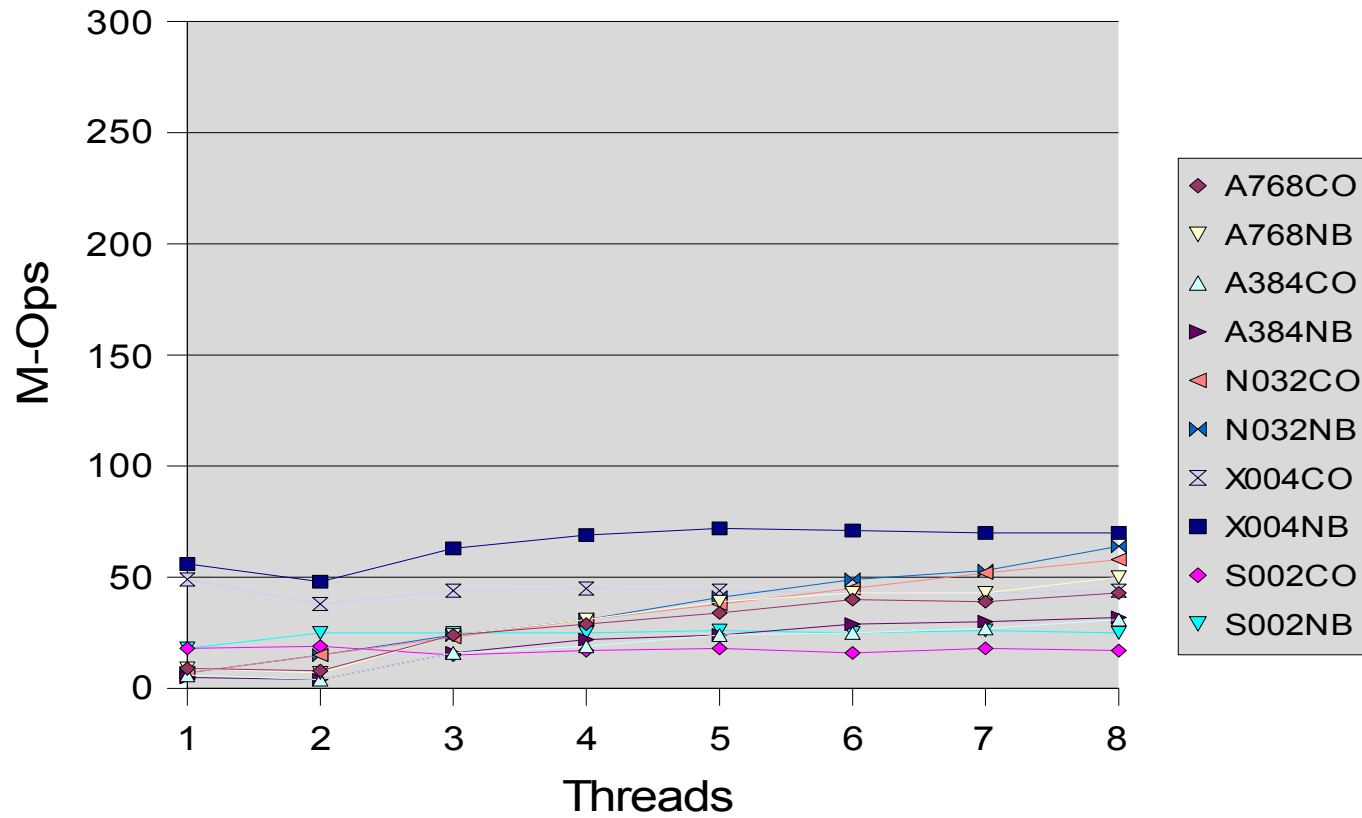
90% Reads

90% Reads



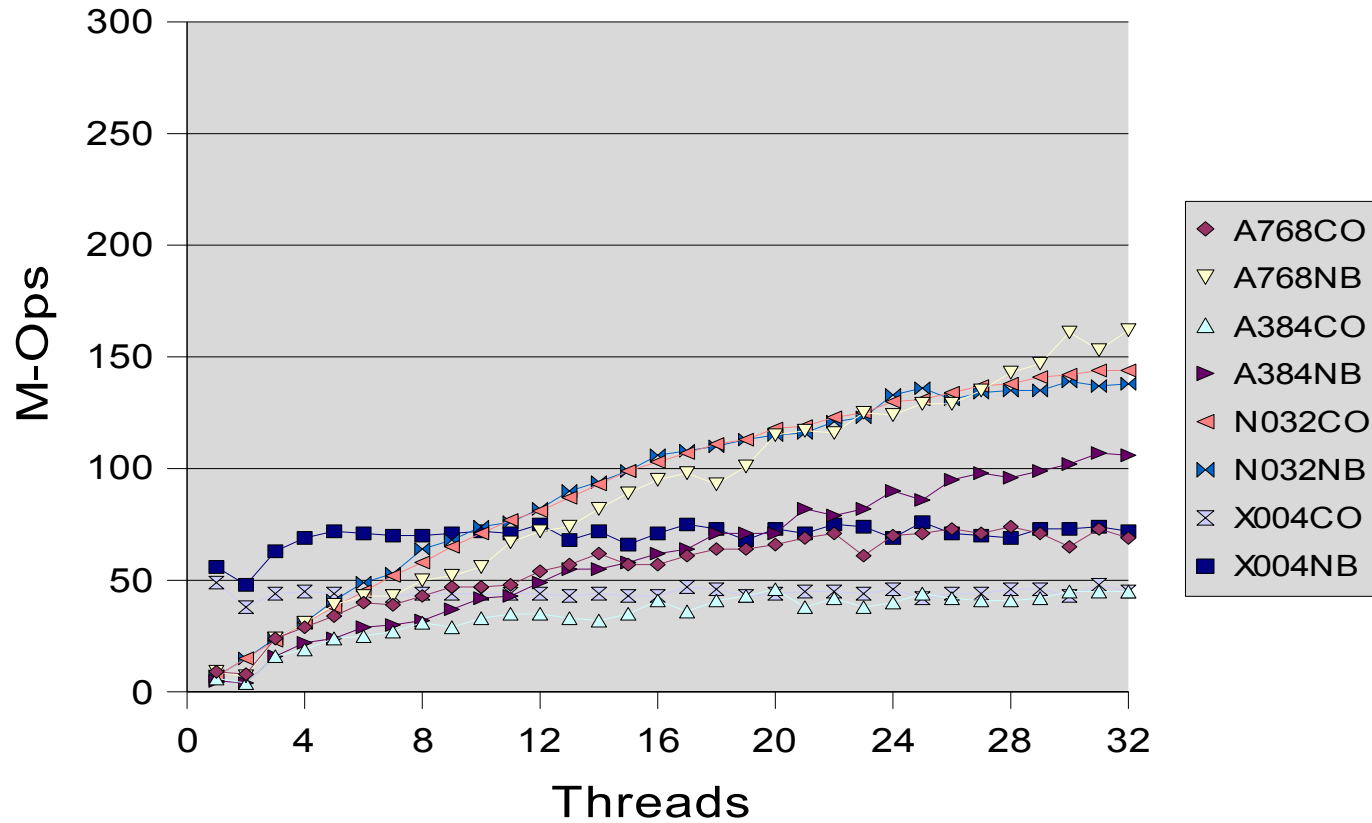
75% Reads

75% Reads



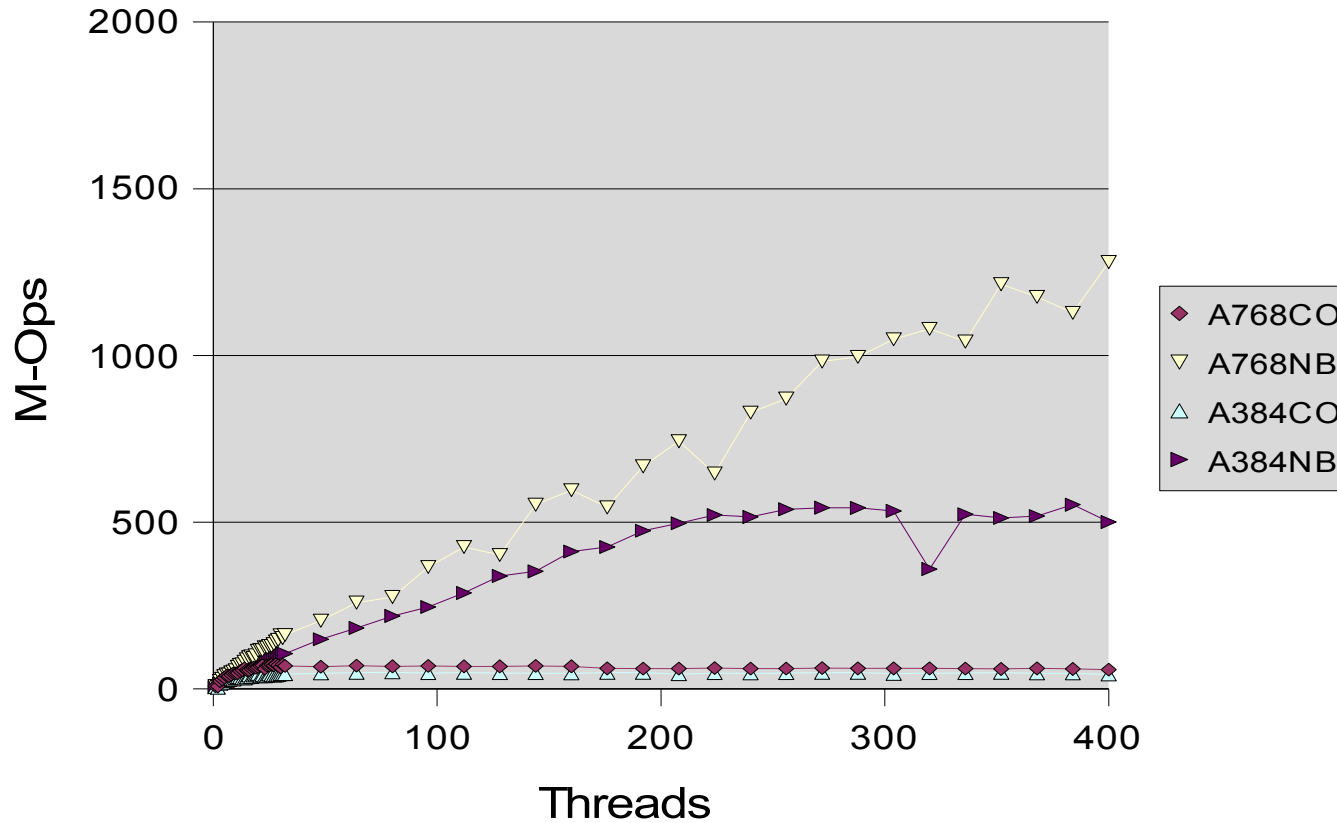
75% Reads

75% Reads

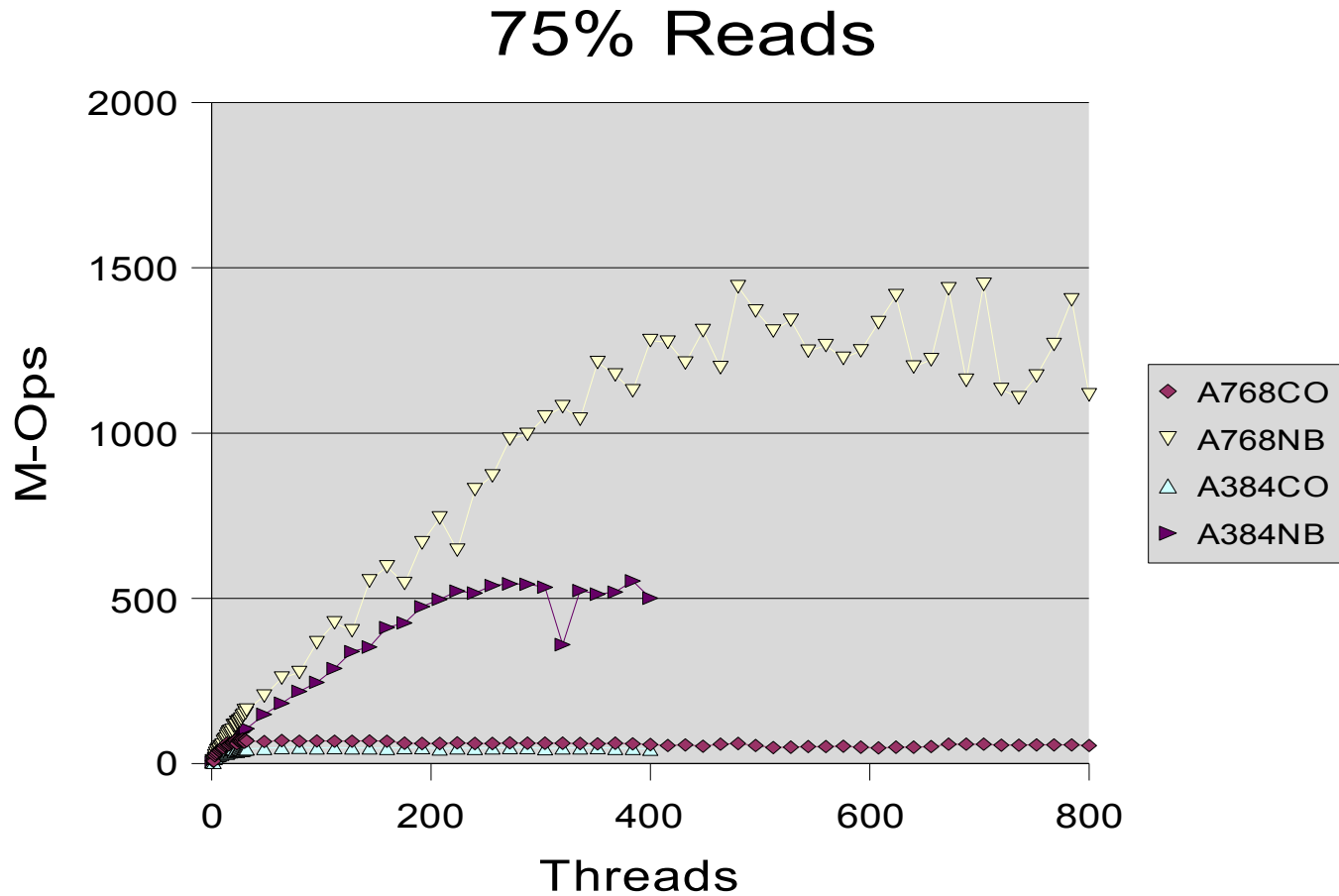


75% Reads

75% Reads



75% Reads



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
- Have a concurrent `j.u.Vector` in the works
- Seems applicable to other data structures as well



UNSHACKLE THE POWER OF JAVA

*NETWORK ATTACHED PROCESSING
FROM AZUL SYSTEMS*



Thank you.
cliffc@acm.org

“Uninteresting” Details

- Could use prime table + MOD
 - Better hash spread, fewer reprobates
 - 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
- Could put Key/Value/Hash on same cache line
- Other variants possible, interesting

“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

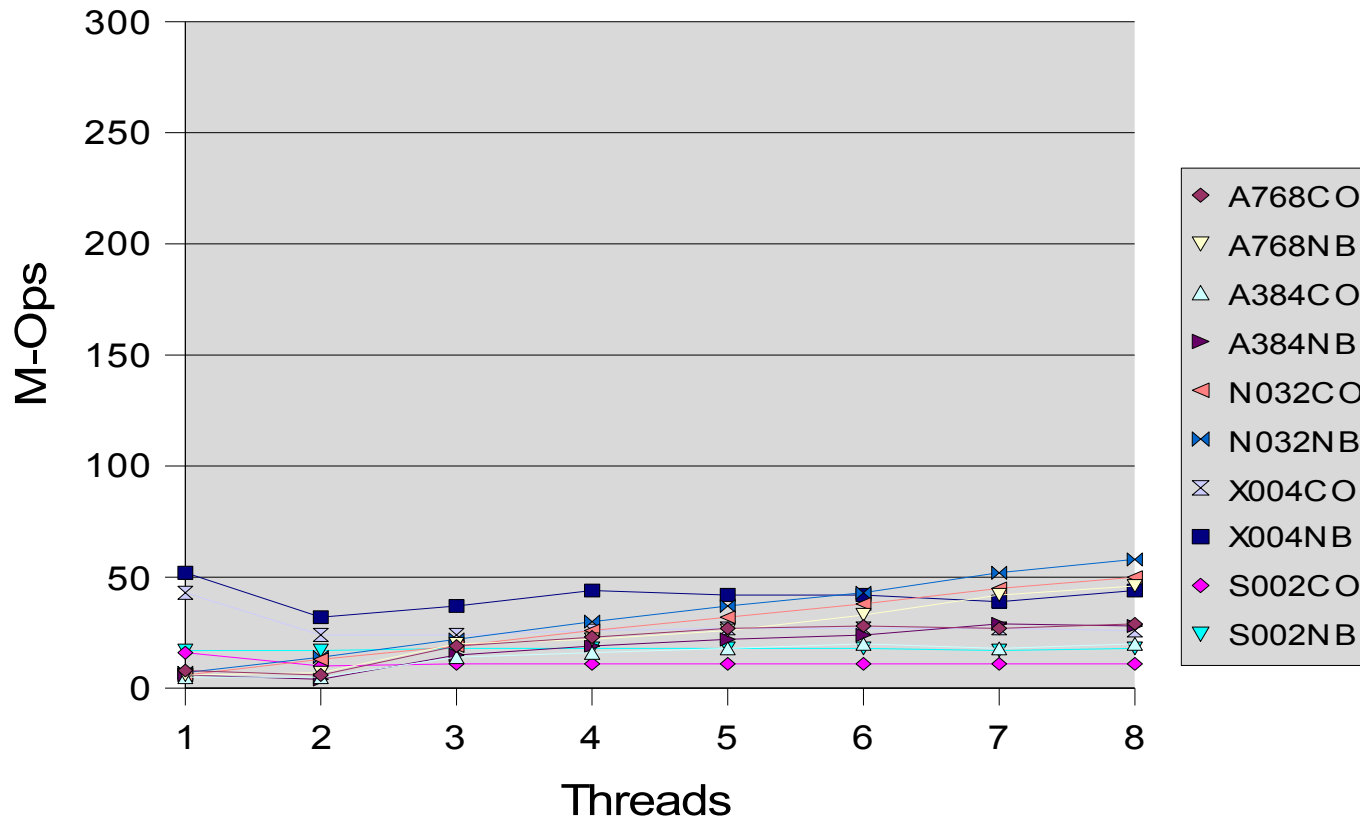
- Requires “no spurious failure” CAS
- No CAS spin-loops
 - Lest you wait forever for success
- Try CAS once
 - If fails – must be contention
 - i.e., Another racing writer is writing
 - Allow other writer to win
- “As If” this write succeeded but was immediately overwritten by another racing writer

Obstruction-Free

- Obstruction-Free: no thread stalled forever
- But resize may never complete:
 - Throbbing in old table can prevent copy
 - Copy attempts slows down table by $O(1)$
 - Requires writers do 'put' at memory bandwidth

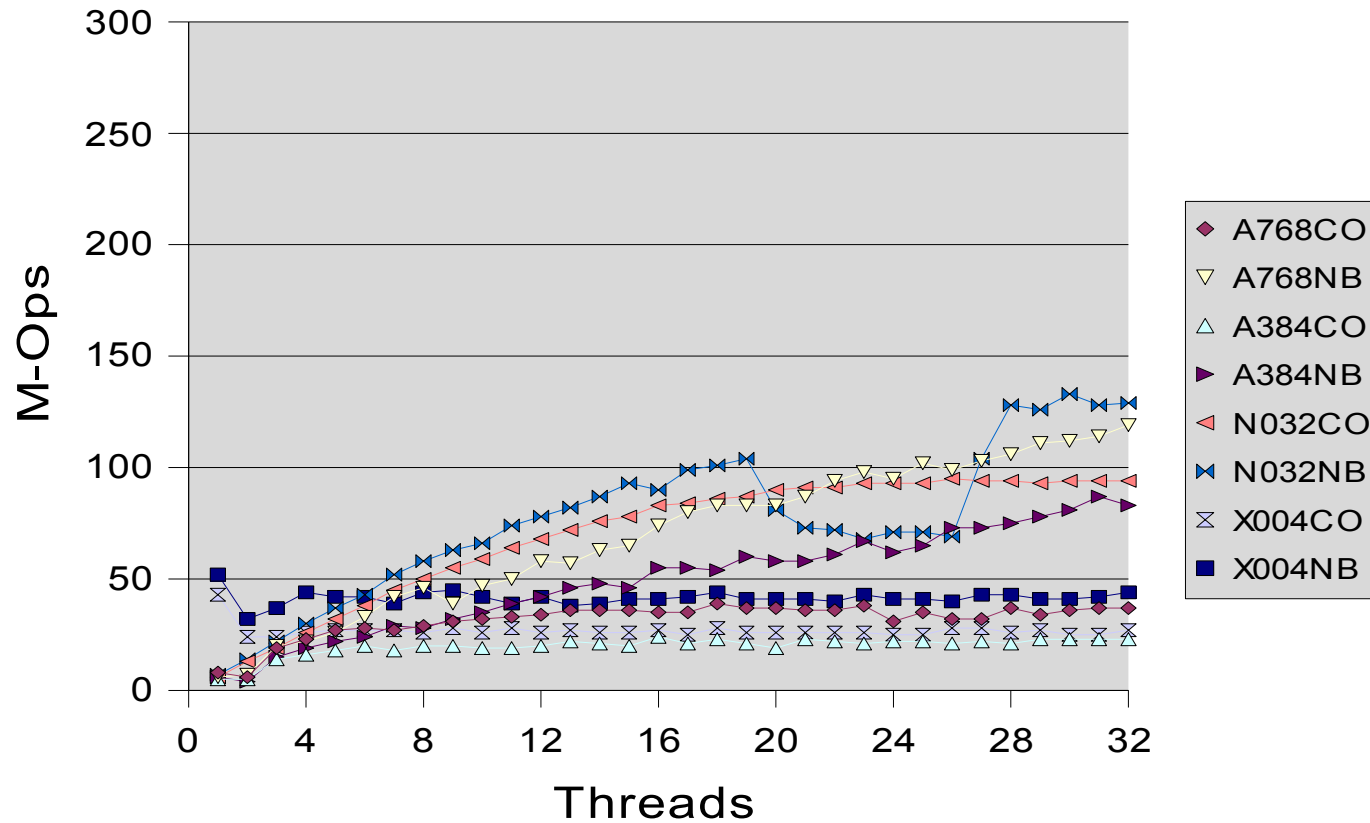
50% Reads

50% Reads

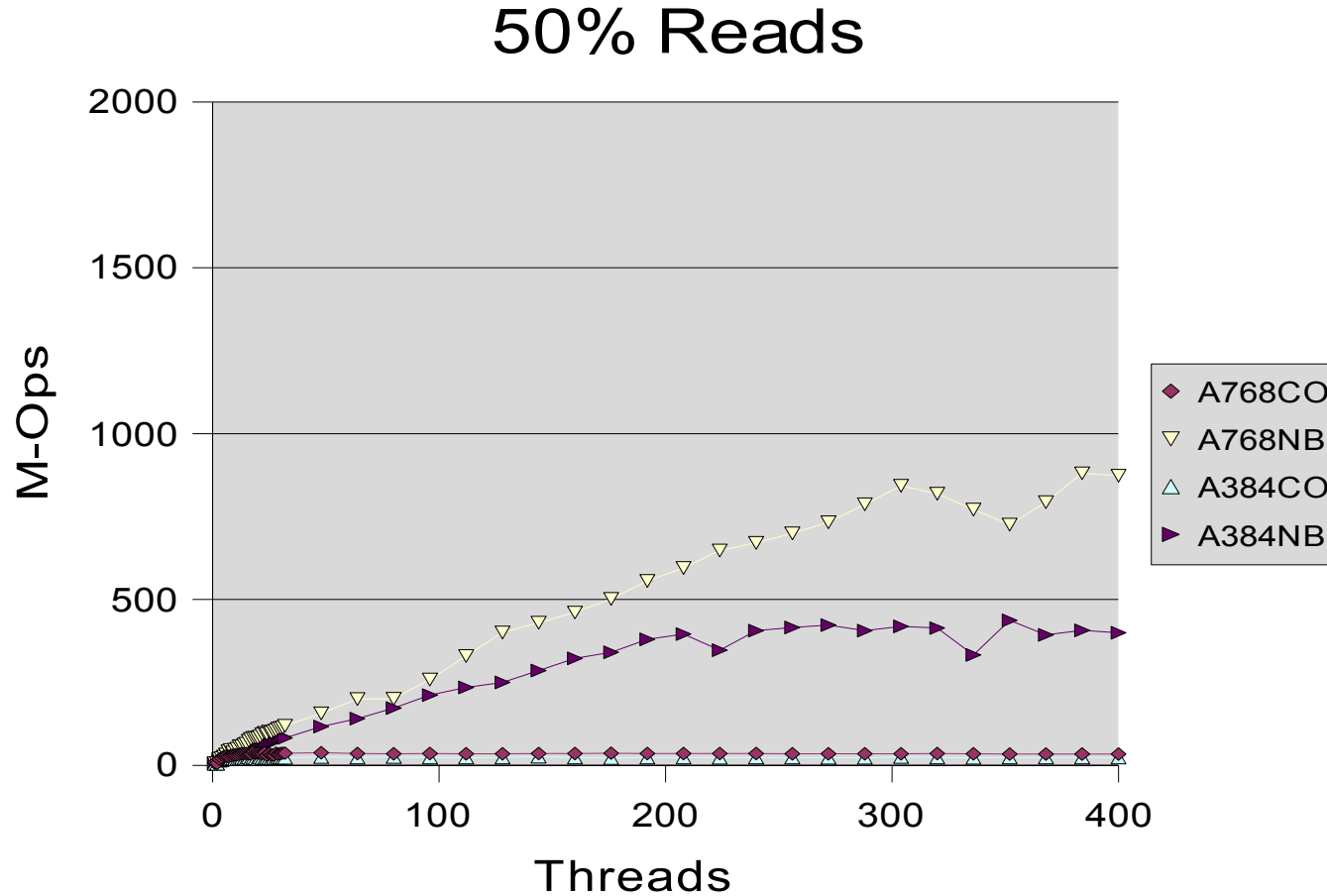


50% Reads

50% Reads



50% Reads



50% Reads

