A Fast Lock-Free Hash Table

Dr Cliff Click

Distinguished Engineer
Azul Systems
blogs.azulsystems.com/cliff

TS-2862

Think Concurrently!

A *Fast* Non-Blocking Hash Table

A Highly Scalable Hash Table

Another way to Think about Concurrency

Agenda

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

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™ Platform Implementations

- Java Platform’s HashTable
  - Single threaded; scaling bottleneck
- HashMap
  - Faster but NOT multi-thread safe
- java.util.concurrent.ConcurrentHashMap
  - 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 Lock-Free Hash Table

- No locks, even during table resize
  - No spin-locks
  - No blocking while holding locks
  - All CAS spin-loops bounded
  - Make progress even if other threads die…
- Requires atomic update instruction:
  - CAS (Compare-And-Swap)
    LL/SC (Load-Linked/Store-Conditional, PPC only), or similar
- Uses sun.misc.Unsafe for CAS

A Faster Hash Table

- Slightly faster than j.u.c for 99% reads < 32 CPUs
- Faster with more CPUs (2x faster)
  - Even with 4096–way striping
  - 10x faster with default striping
- 3x Faster for 95% reads (30x vs default)
- 8x Faster for 75% reads (100x vs default)
- Scales well up to 768 CPUs, 75% reads
  - Approaches hardware bandwidth limits

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 platform 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 and value on same cache line
- Hash memoized
  - Should be same cache line as K + V
  - But hard to do in pure Java code
- No allocation on get() or put()
- Auto-resize

Example get() Code

```
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);        // get 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

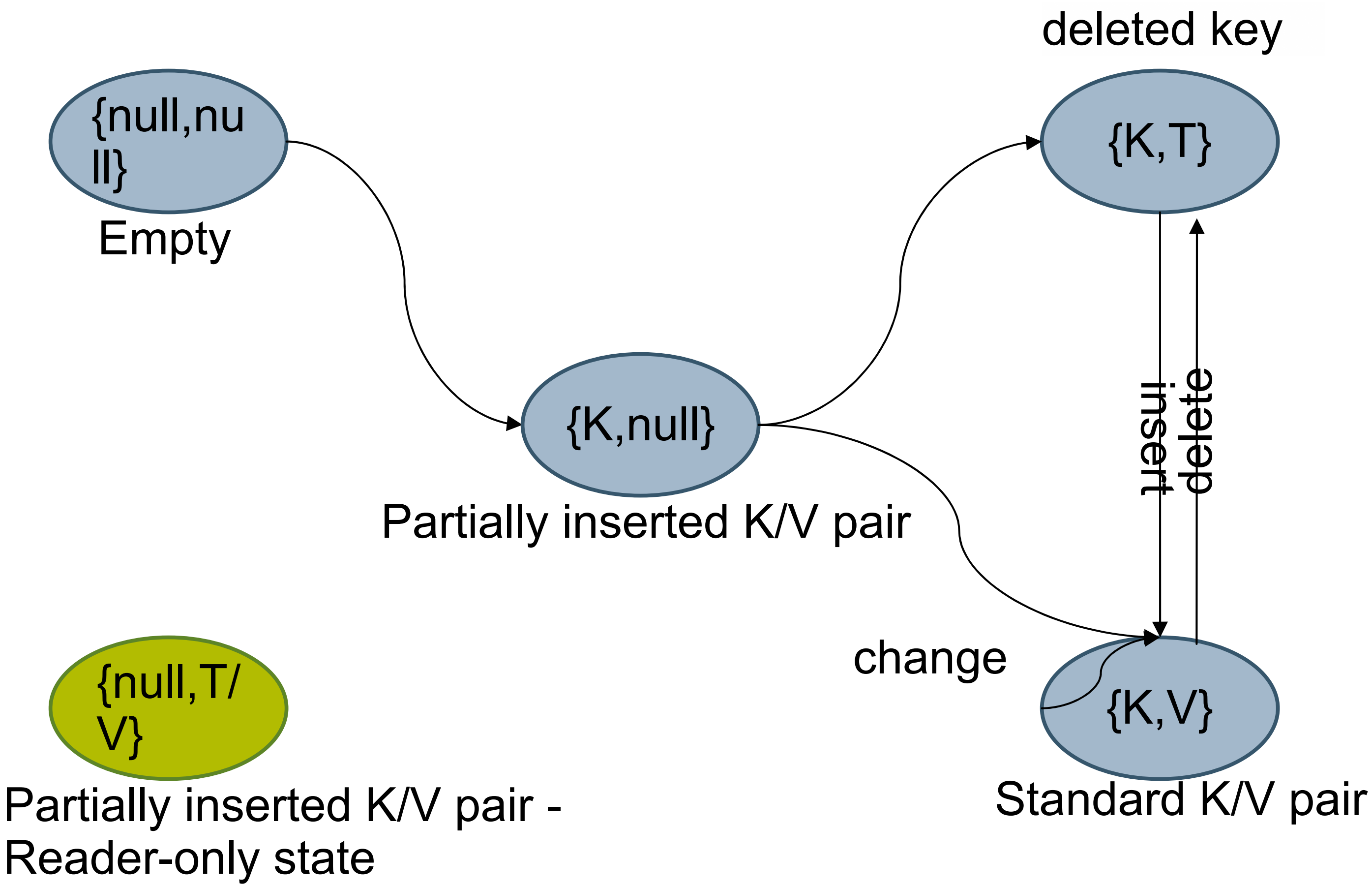
State-Based Reasoning

- 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:
  - null—empty
  - K—some Key; can never change again
- A Value slot is:
  - null—empty
  - T—tombstone, for deleted values
  - V—some Values
- A state is a {Key,Value} pair
- A transition is a successful CAS

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 {null,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 and writers, no synchronization
  - i.e., darned little

Example put(key,newval) Code

```
idx = hash = key.hashCode();
while( true ) {                        // Key-Claim stanza
   idx &= (size-1);
   k = get_key(idx);                   // State: {k,?}
   if( k == null &&                        // {null,?} -> {key,?}
       CAS_key(idx,null,key) )
     break;                            // State: {key,?}
   h = get_hash(idx);                  // get memoized hash
   if( k == key || (h == hash && key.equals(k)) )
     break;                            // State: {key,?}
   idx++;                              // reprobe
}
```

Example put(key,newval) Code

```
// State: {key,?}
oldval = get_val(idx);  // State: {key,oldval}
// Transition: {key,oldval} -> {key,newval}
if( CAS_val(idx,oldval,newval) ) {
  // Transition worked
  ...                     // Adjust size
} else {
  // Transition failed; oldval has changed
  // We can act "as if" our put() worked but
  // was immediately stomped over
}
return oldval;
```

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 and 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 and 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

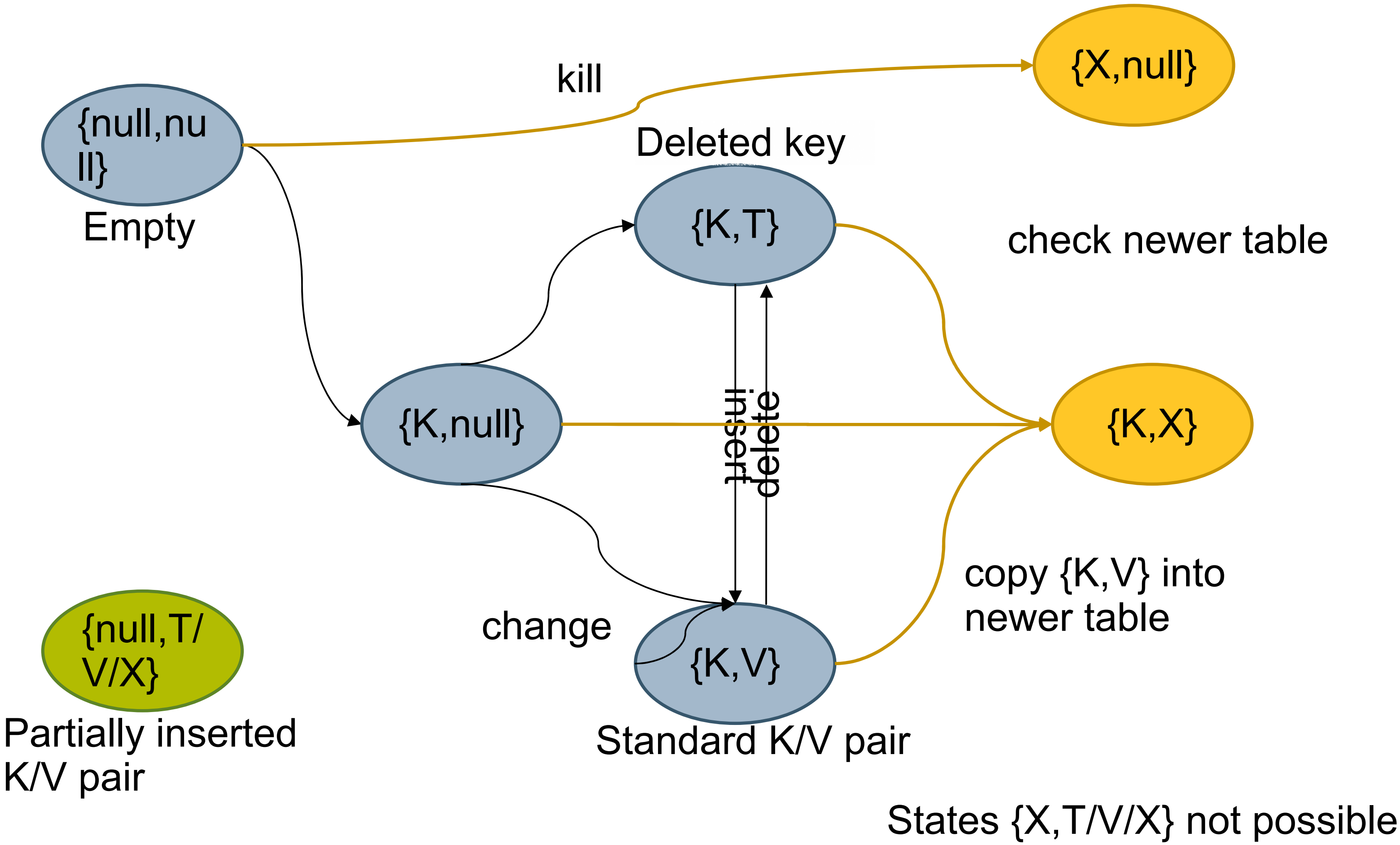
get/put During Resize

- 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
- Copying K/V pairs is independent of get/put
- Copy has many heuristics to choose from:
  - All touching threads, only writers, unrelated background thread(s), etc

New State: “use new table” Sentinel

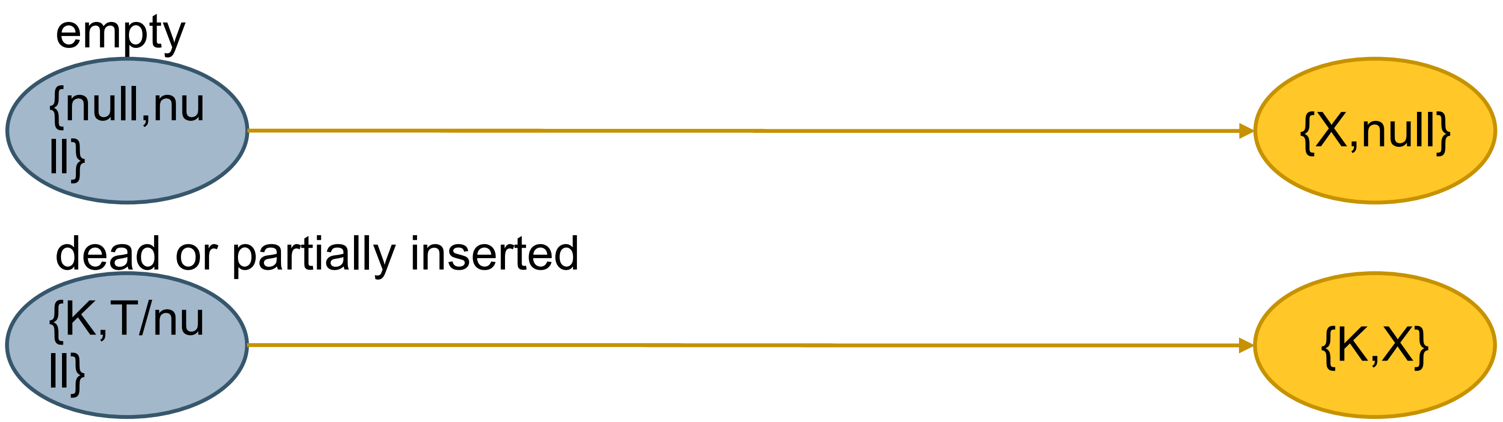
- X: Sentinel used during table-copy
  - Means: not in old table, check new
- A Key slot is:
  - null, K
  - X—“use new table”, not any valid key
  - null → K OR null → X
- A value slot is:
  - null, T, V
  - X—“use new table”, not any valid Value
  - null → {T,V}* → X

State Machine—Old Table

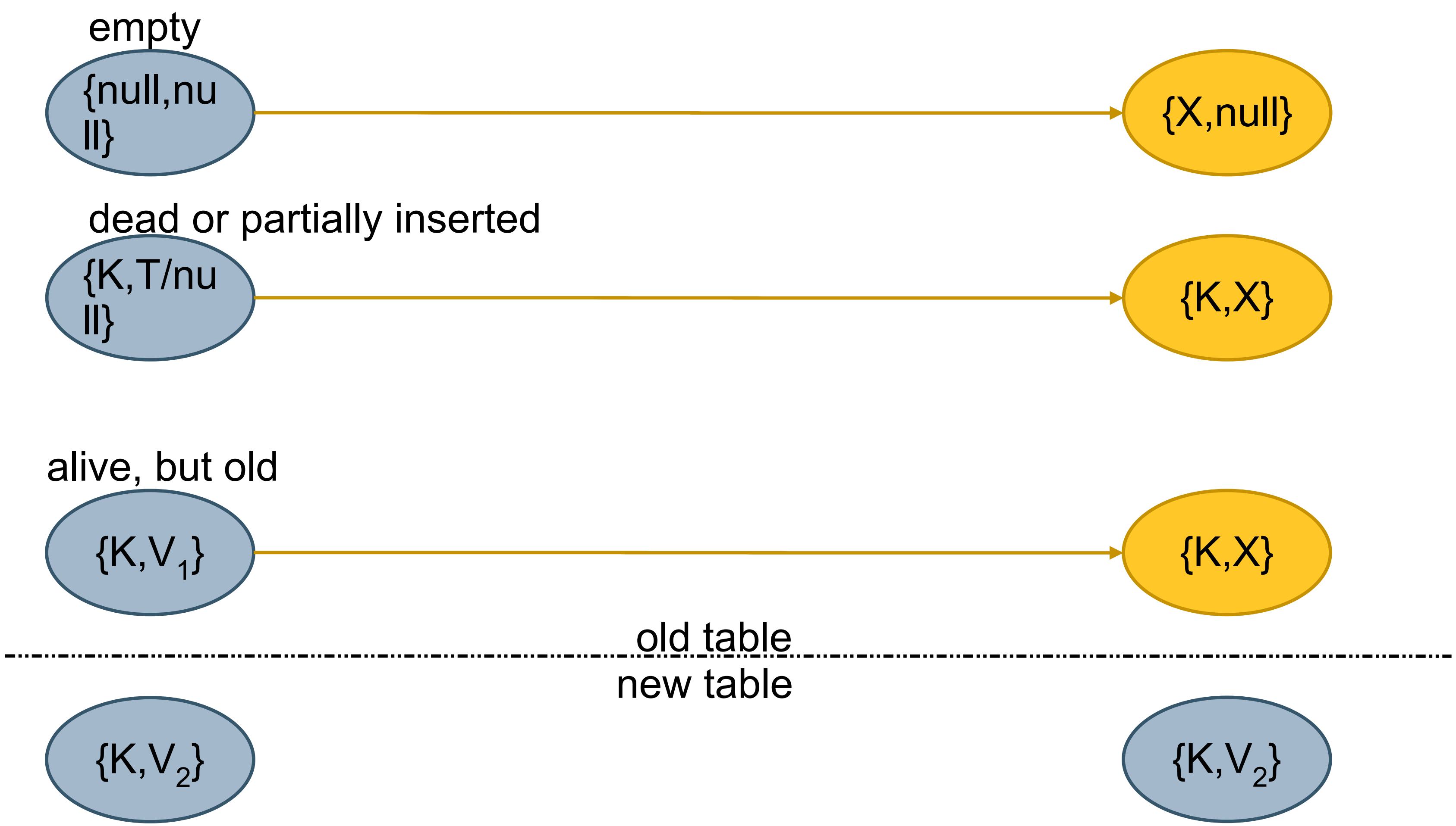


State Machine: Copy One Pair

State Machine: Copy One Pair



State Machine: Copy One Pair



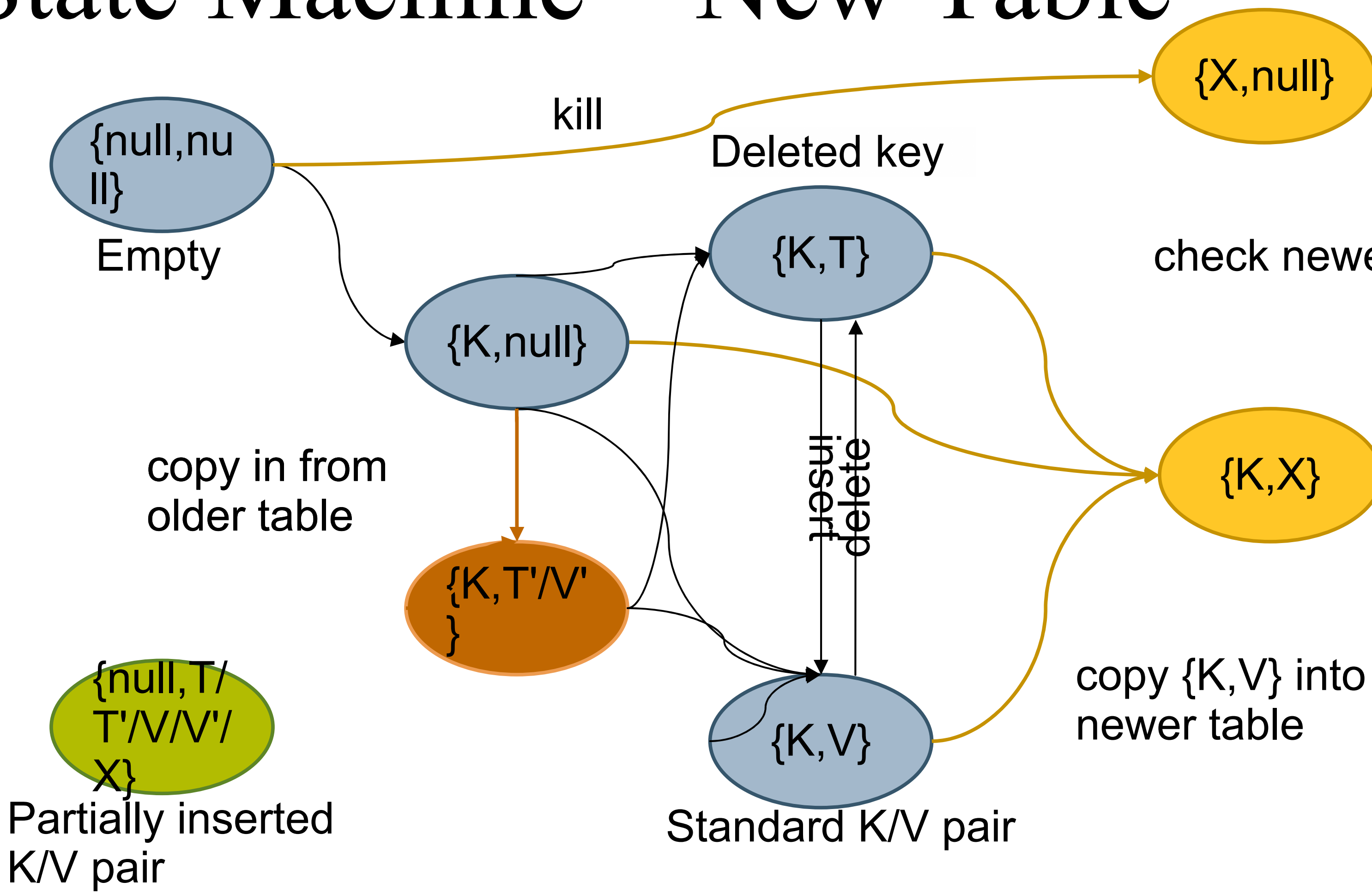
Copying Old to New

- New States V', T'—primed versions of V,T
  - Prime'd values in new table copied from old
  - Non-prime in new table is recent put()
  - "happens after" any prime'd value
  - Engineering: wrapper class, steal a bit (C)
- Must be sure to copy late-arriving old-table write
- Attempt to copy atomically
  - May fail and copy does not make progress
  - But old, new tables not damaged
- Prime allows 2-phase commit

New States: Prime'd

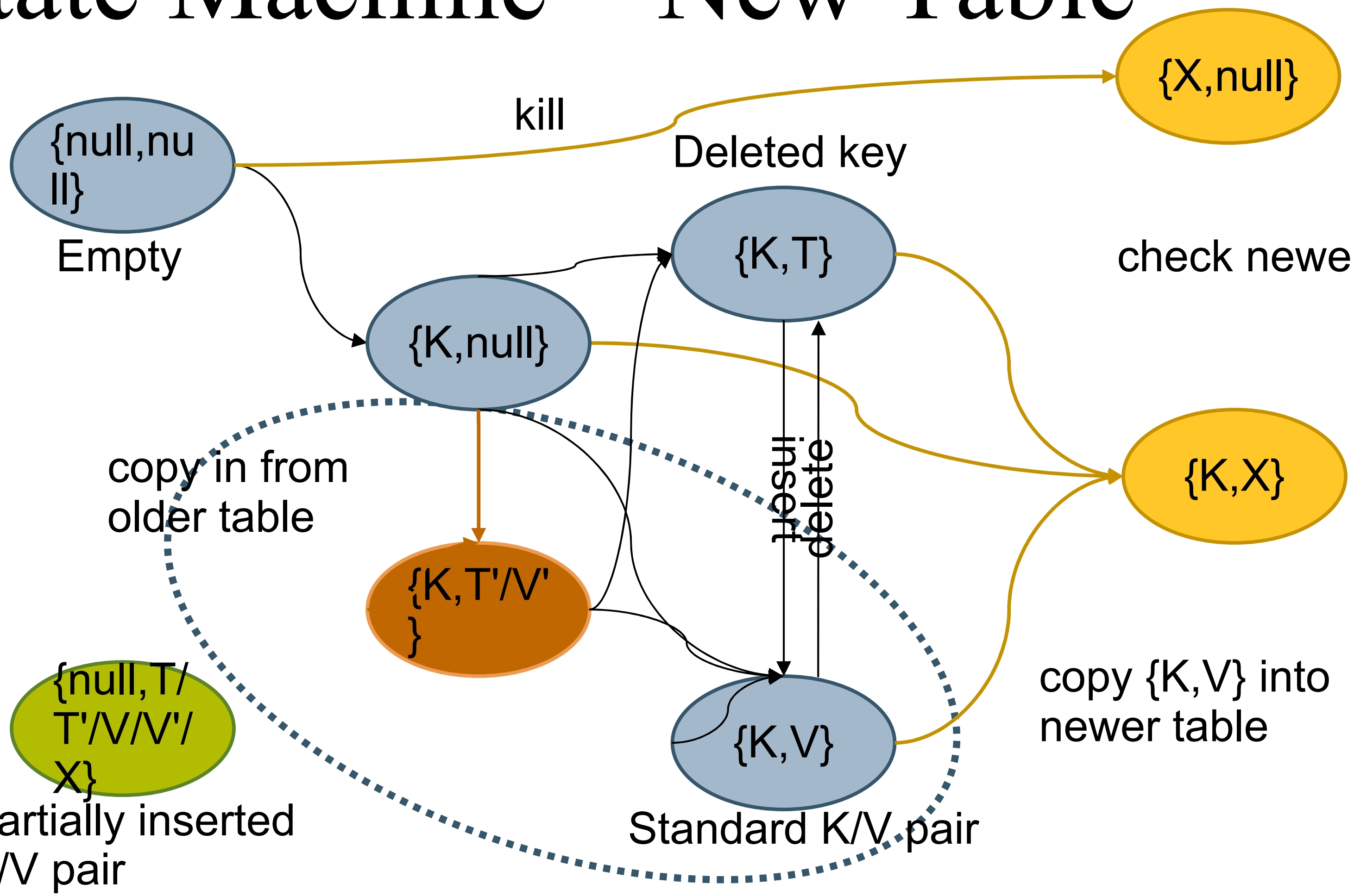
- A Key slot is:
  - null, K, X
- A Value slot is:
  - null, T, V, X
  - T',V' – primed versions of T and V
  - Old things copied into the new table
  - "2-phase commit"
  - null → {T',V'}* → {T,V}* → X
- State machine again…

State Machine—New Table



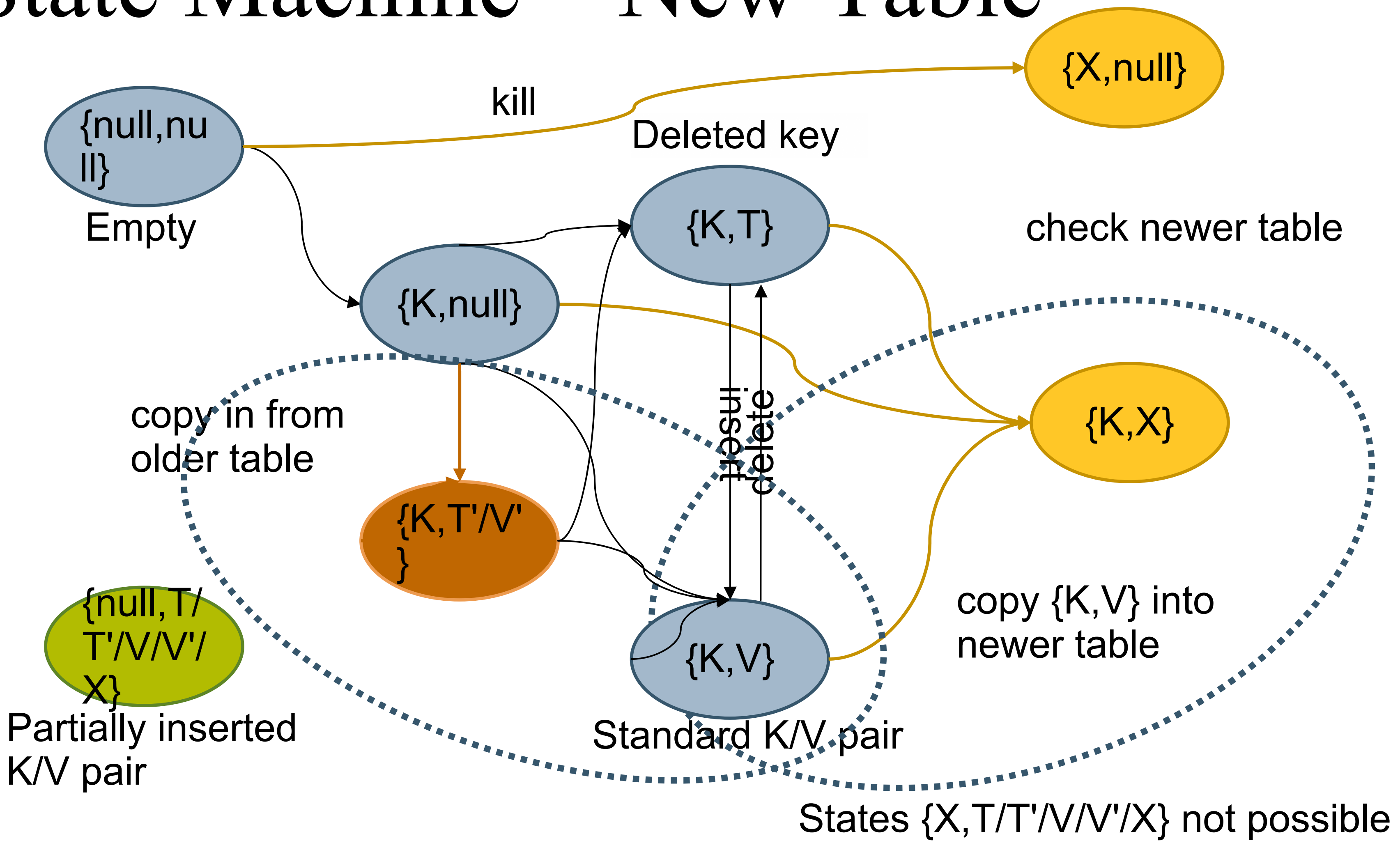
States {X,T/T'/V/V'/X} not possible

State Machine—New Table

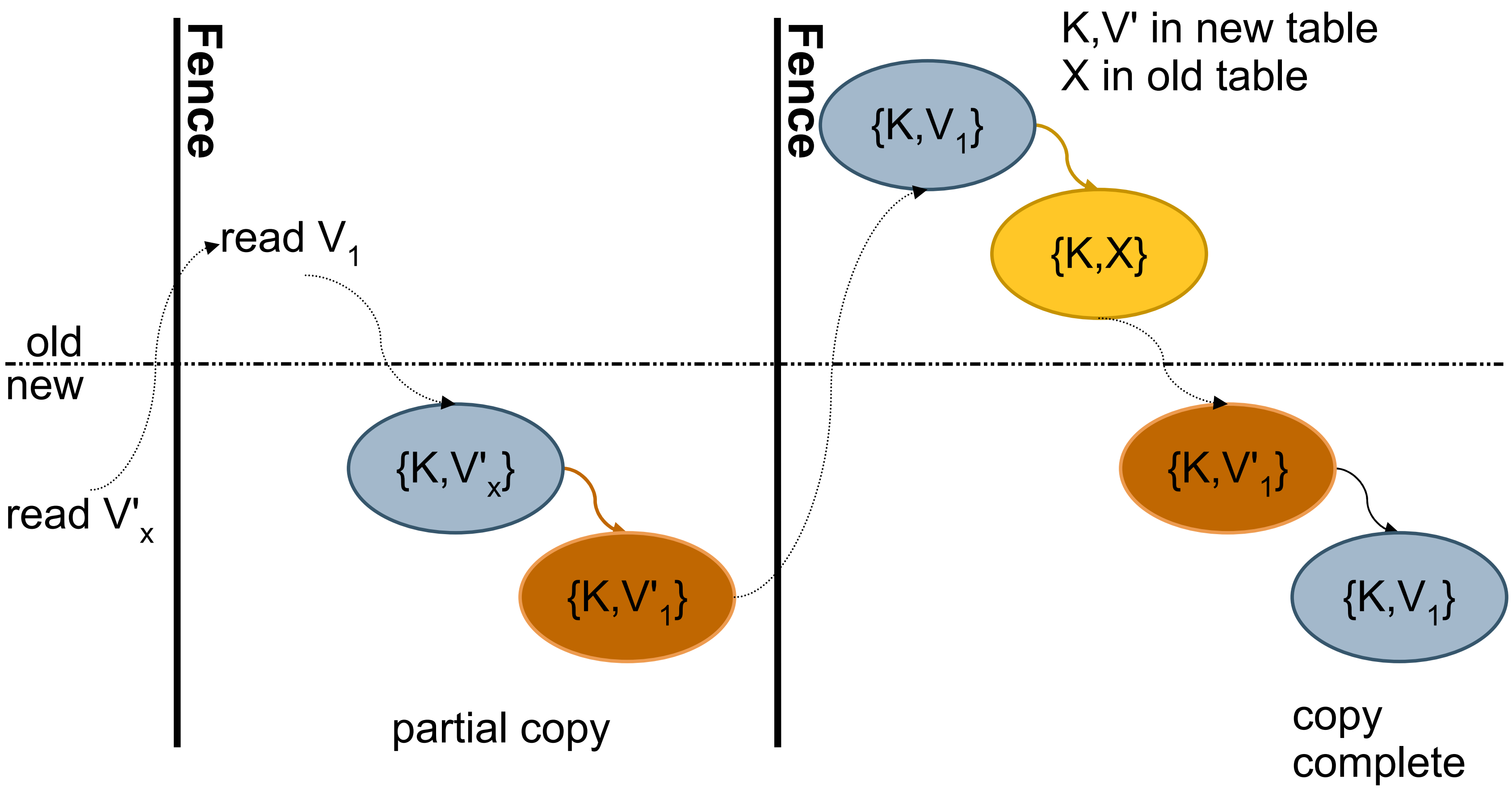


States {X,T/T'/V/V'/X} not possible

State Machine—New Table



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)
- 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
- **Any** thread can see **any** state at **any** time
  - And CAS to the next state

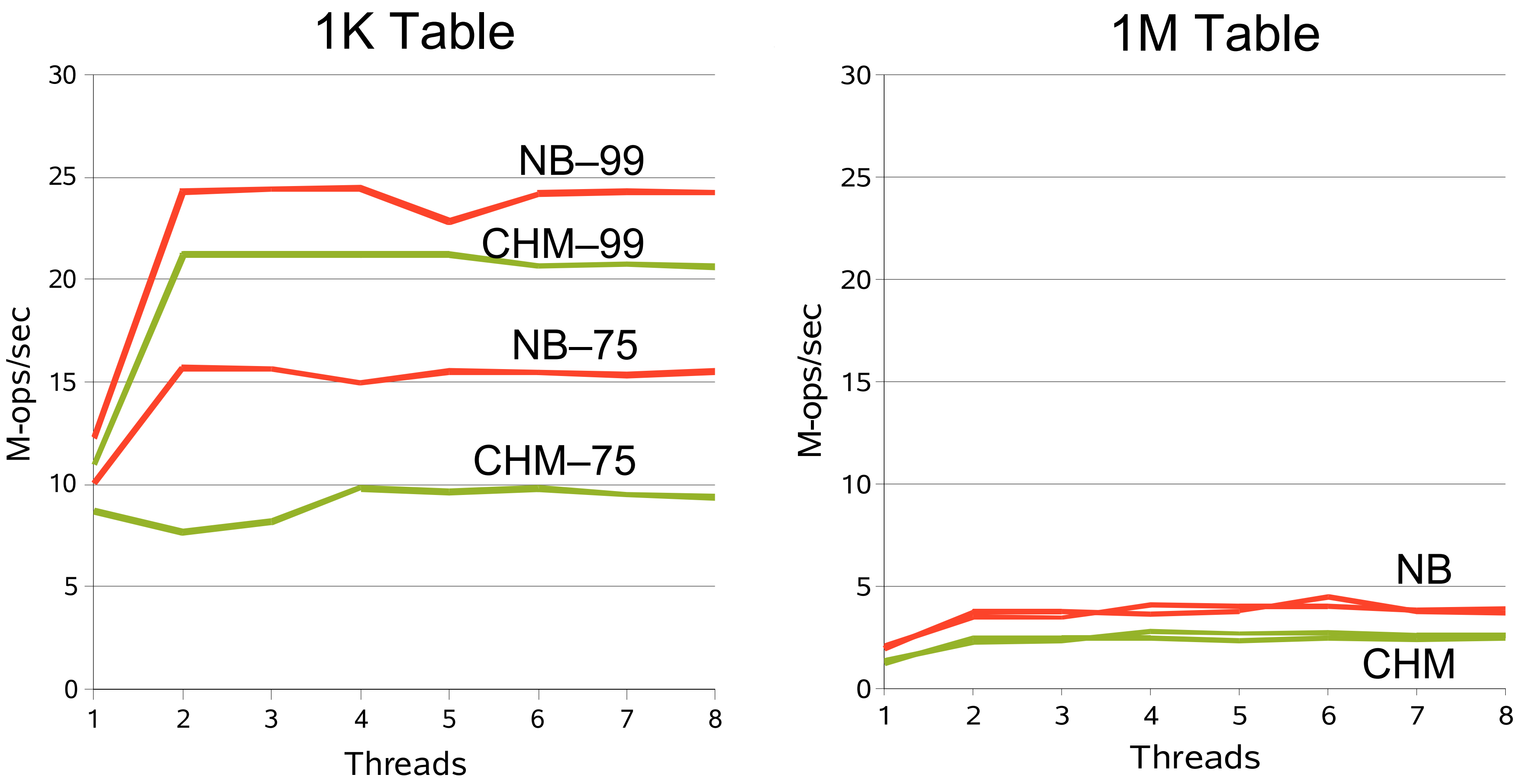
Agenda

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

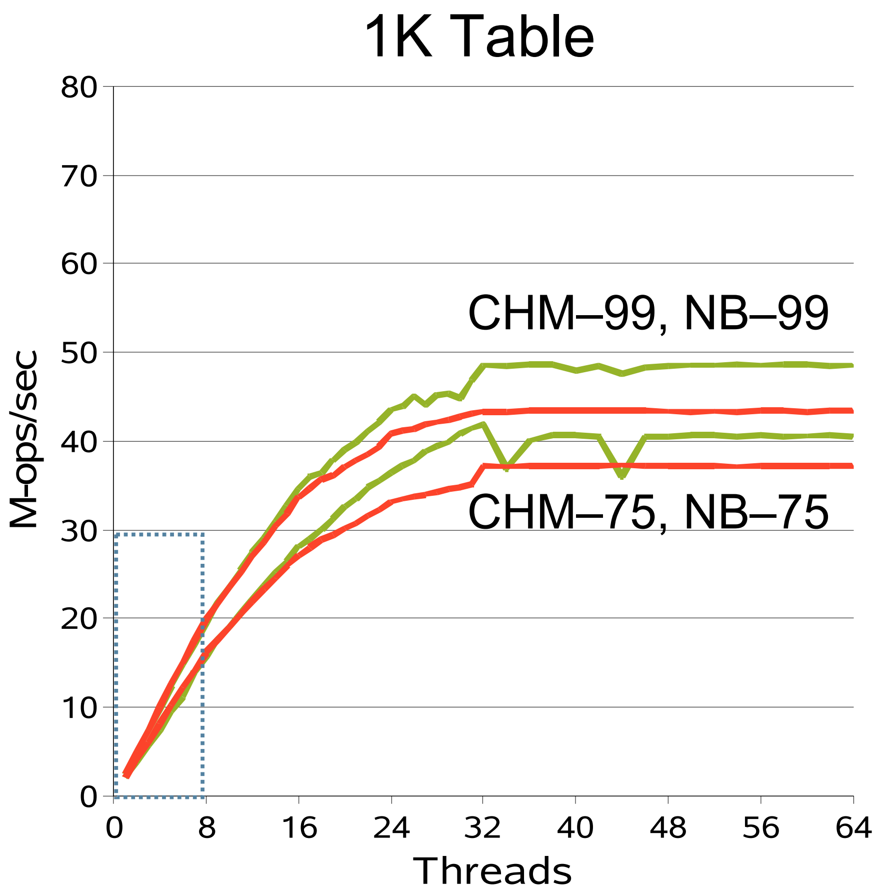
Microbenchmark

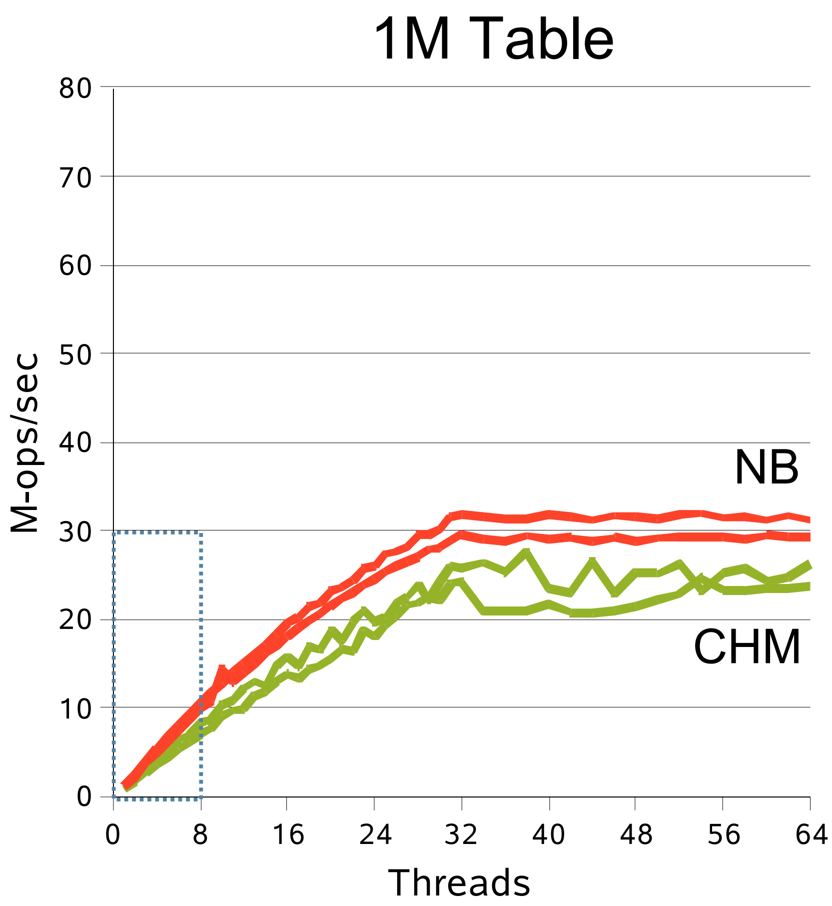
- Measure insert/lookup/remove of strings
- Tight loop: No work beyond HashTable itself and test harness (mostly RNG)
- "Guaranteed not to exceed" numbers
- All fences; full ConcurrentHashMap semantics
- Variables:
  - 99% get, 1% put (typical cache) vs 75/25
  - Dual Athalon, Niagara, Azul Vega1, Vega2
  - Threads from 1 to 800
  - NonBlocking vs 4096-way ConcurrentHashMap
  - 1K entry table vs 1M entry table

AMD 2.4Ghz—2(HT) CPUs

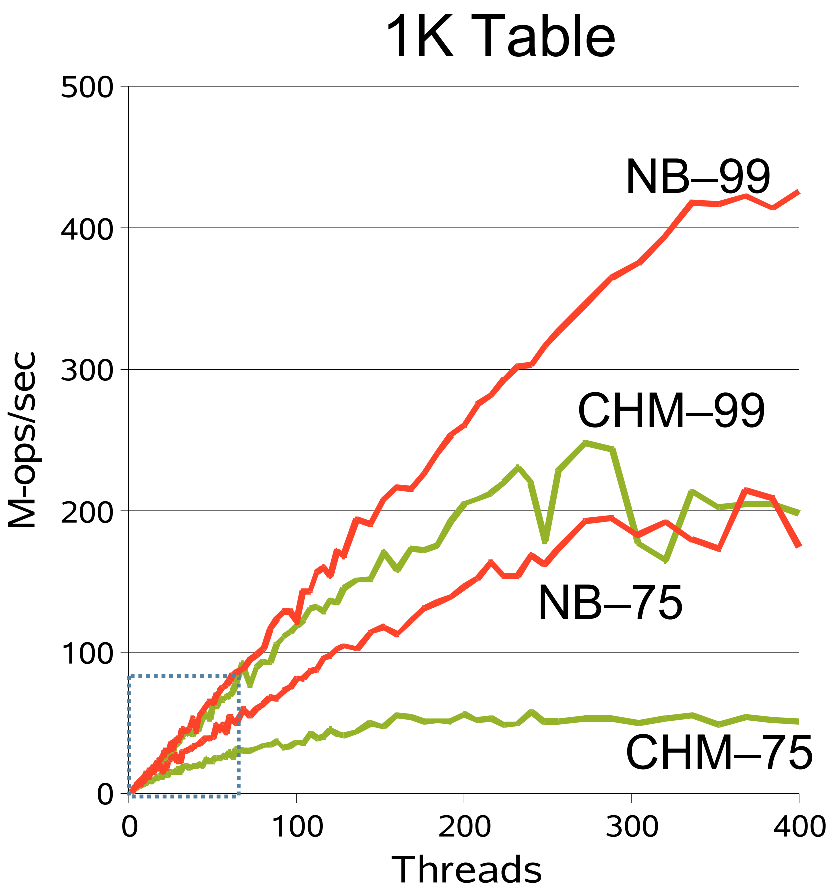


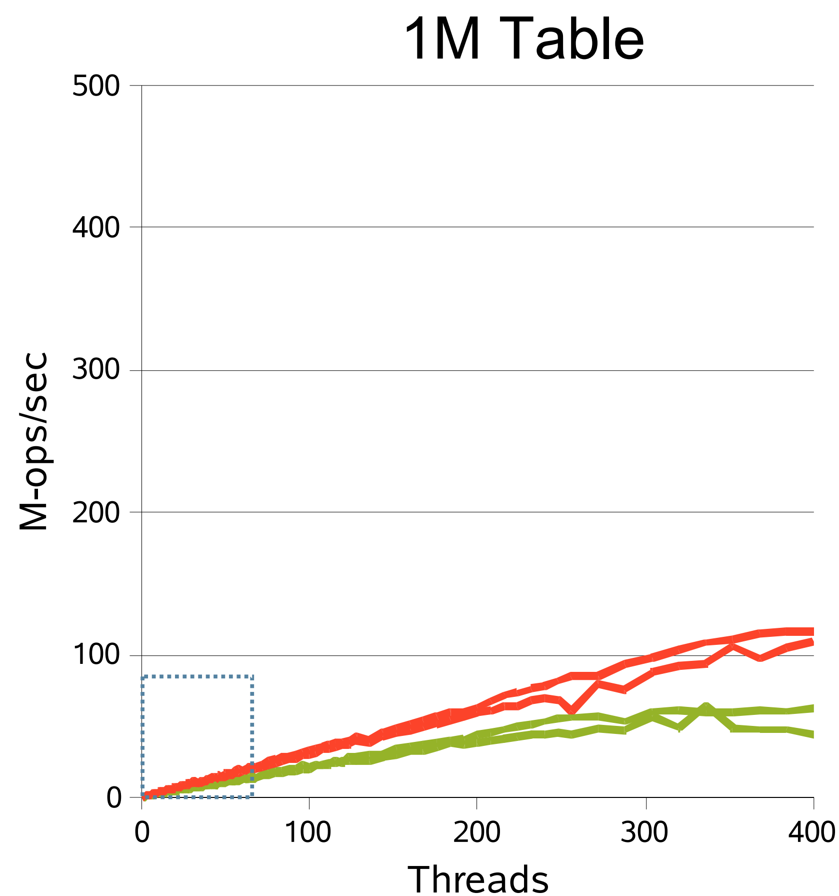
Niagara—8x4 CPUs



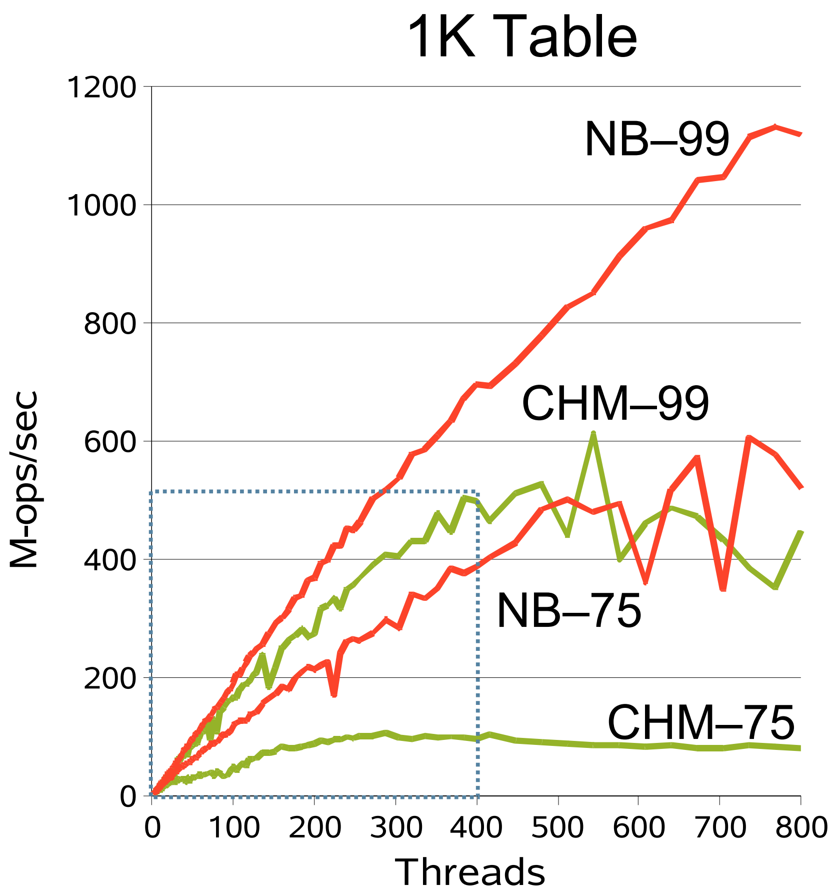


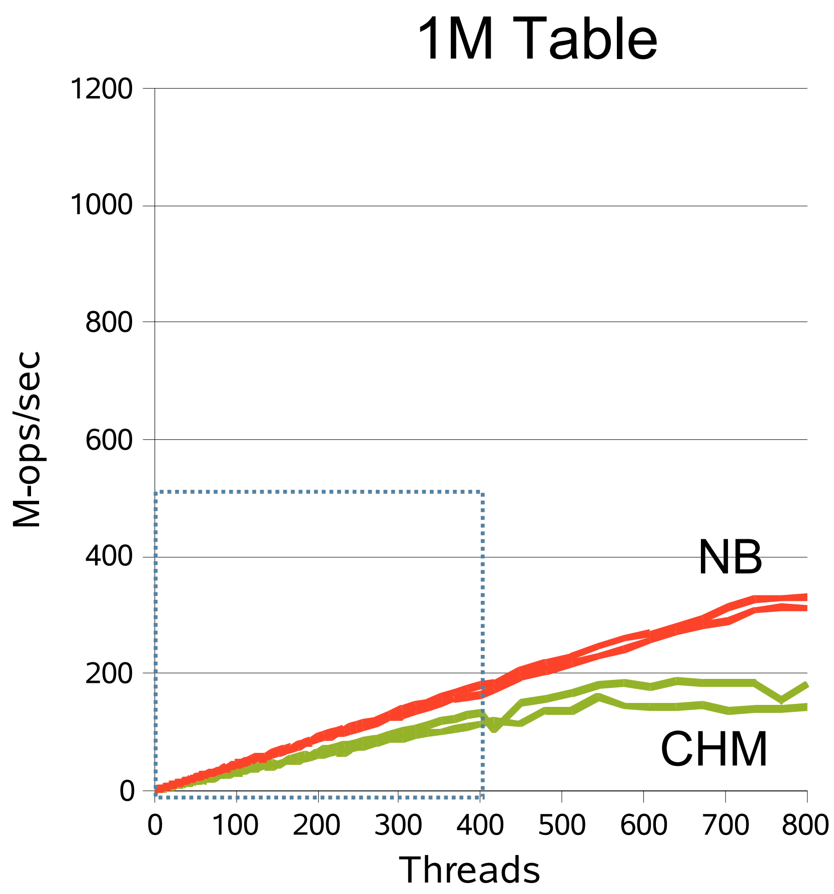
Azul Vega1—384 CPUs





Azul Vega2—768 CPUs





Summary

- A faster lock-free HashTable
- Faster for more CPUs
- Much faster for higher table modification rate
- State-Based Reasoning:
  - No ordering, no JMM, no fencing
- **Any** thread can see **any** state at **any** time
  - Must assume values change at each step
- State graphs **really** helped coding and debugging
- Resulting code is small and fast

Summary

- Obvious future work:
  - Tools to check states
  - Tools to write code
- Seems applicable to other data structures as well
- Code available at:
  - https://sourceforge.net/projects/high-scale-lib
- See also TS-2220,
  Testing Concurrent Software
  - http://www.azulsystems.com/blogs/cliff/

Q&A

A Fast Lock-Free Hash Table

Dr Cliff Click

Distinguished Engineer
Azul Systems
blogs.azulsystems.com/cliff

TS-2862