Userspace RCU Library: RCU Synchronization and RCU/Lock-Free Data Containers for Userspace

E-mail:
mathieu.desnoyers@efficios.com

> Presenter

- Mathieu Desnoyers
- EfficiOS Inc.
  - http://www.efficios.com
- Author/Maintainer of
  - LTTng, LTTng-UST, Babeltrace, LTTV, Userspace RCU

> Outline

- Userspace RCU
- Data structures
- User-space wake-up management

> Userspace RCU

- Initially motivated by the need for a RCU library to perform efficient user-space tracing (LTTng-UST project)
- Provides linear read-side scalability with respect to number of cores.
- Released under LGPL license.

> Userspace RCU (2)

- All RCU flavors keep track of RCU readers on a per-thread basis.
- No interaction with kernel-level scheduler.
- Current implementation requires pthreads for thread management.

> Userspace RCU (3)

- 4 Userspace RCU flavors
  - urcu-mb: memory-barrier based, uses read-side critical section nesting counter. Friendly for library usage.
  - urcu-qsbr: reader threads report quiescent states periodically. Lowest overhead.
  - urcu-signal: similar to urcu-mb, but with lower overhead. Reserves a signal number.
  - urcu based on sys_membarrier (IPI scheme)
    - Low-overhead and library-friendly.
    - Waiting for system call mainlining (***need users***)

> Userspace RCU (4)

- call_rcu support
  - Mechanism to support delayed execution without blocking the caller.
  - Configurable RCU worker threads:
    - Per-thread
    - Per-CPU
    - Global
  - Efficient xchg-based wait-free enqueue to manage call_rcu work.

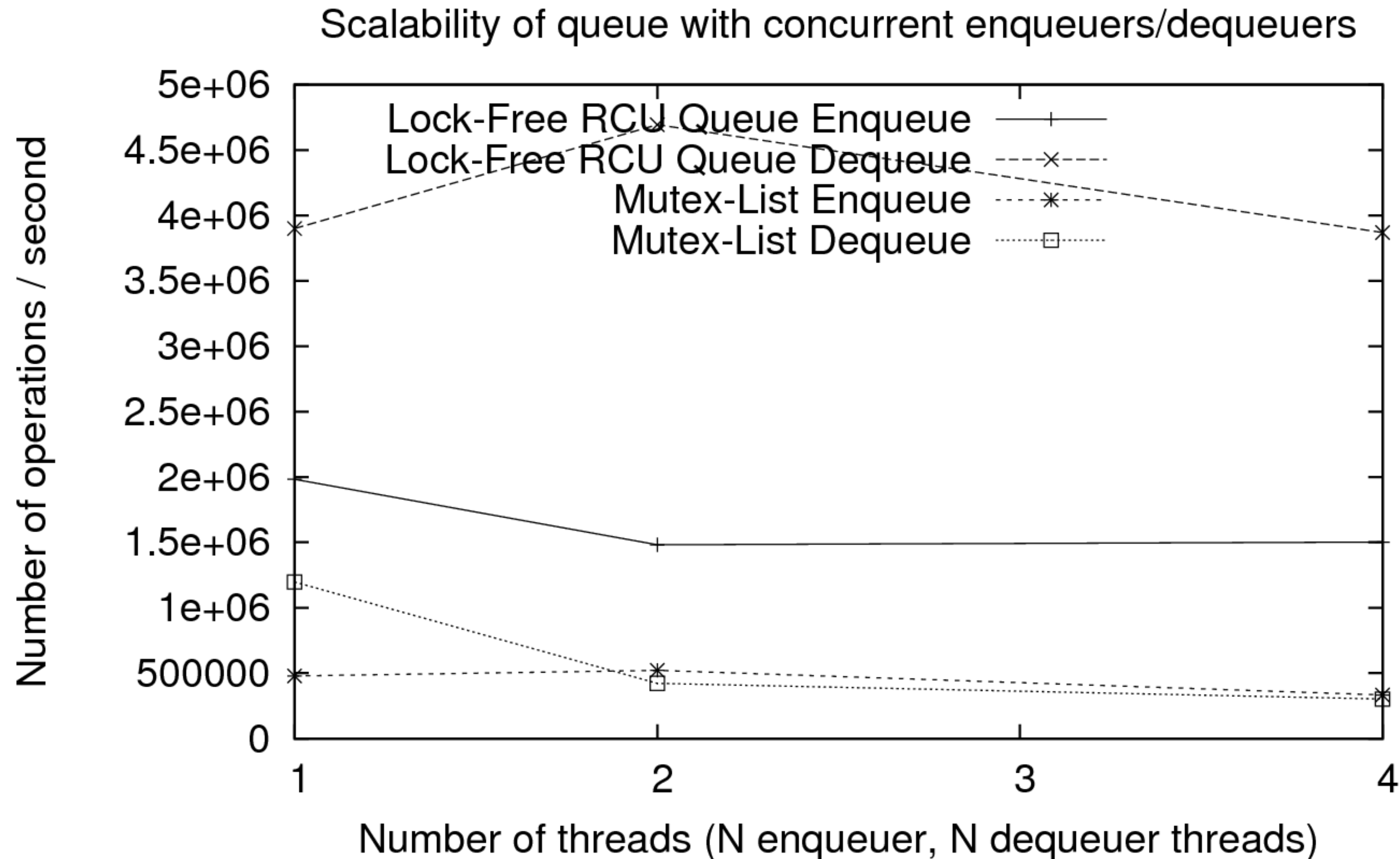
> Data Structures

- Mutex-protected double-linked lists
- RCU lock-free queue
- RCU lock-free stack
- RCU split-ordered lock-free resizable hash table
- RCU red-black tree

> RCU Lock-Free Queue

- RCU read-side for cmpxchg ABA on enqueue and dequeue.
- Allows concurrent enqueue and dequeue by not sharing any cache-line except for the transiting nodes.
- Queue initialized with a dummy node.
- Dequeue allocate a dummy node before dequeuing the last queue node. Dummy nodes are reclaimed internally with call_rcu when dequeued.
- Assumes performance matters mainly when queue has more than 1 element.

> RCU Lock-Free Queue (benchmarks)

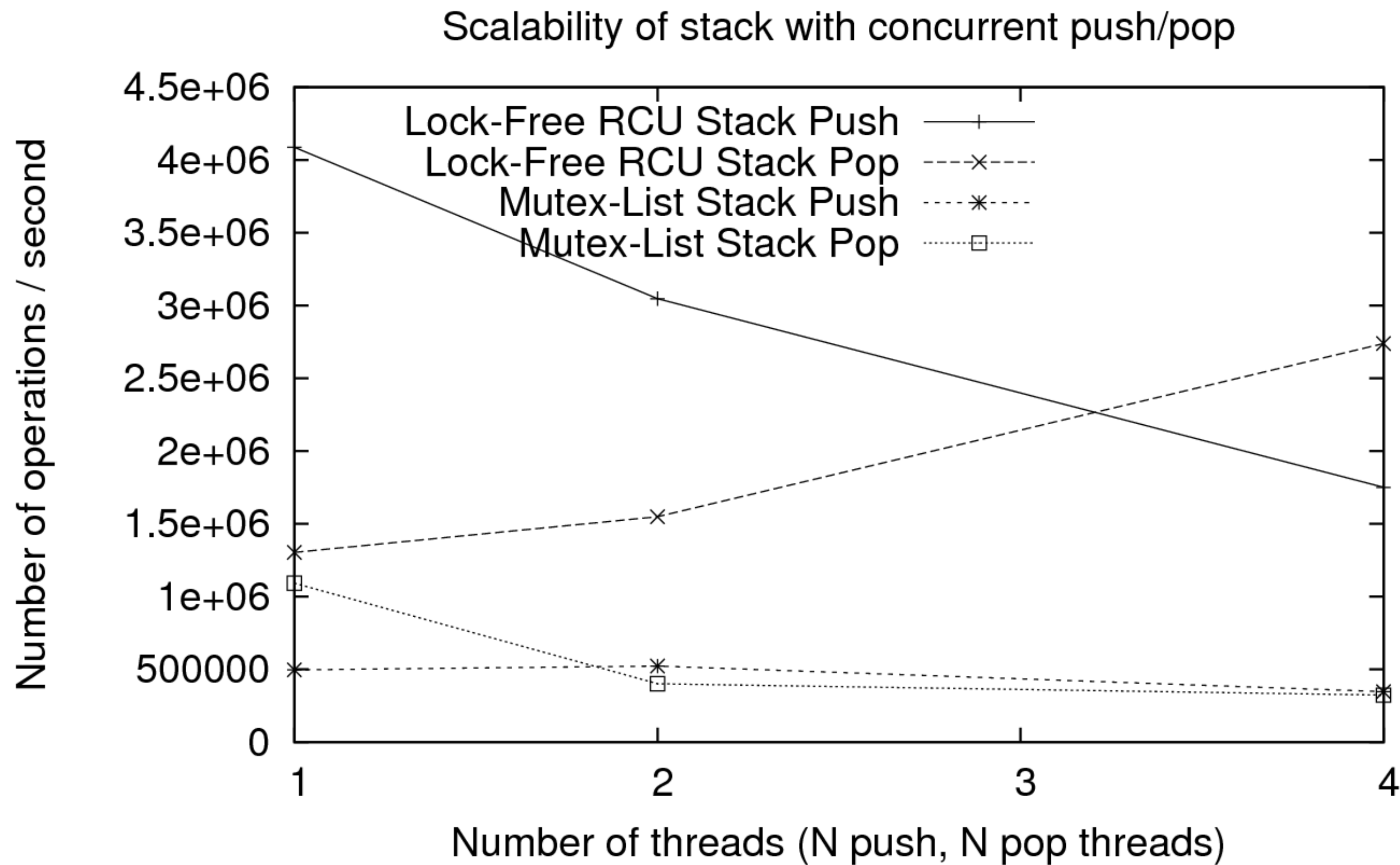


Benchmarks performed on a 2-sockets * 4 core/socket Intel Xeon Core2 2GHz with 16 GB ram.

> RCU Lock-Free Stack

- Uses RCU to deal with cmpxchg ABA on pop.
- Bottom of stack marked with a NULL node.

> RCU Lock-Free Stack (benchmarks)



Benchmarks performed on a 2-sockets * 4 core/socket Intel Xeon Core2 2GHz with 16 GB ram.

> RCU Split-Ordered Lock-Free Resizable Hash Table

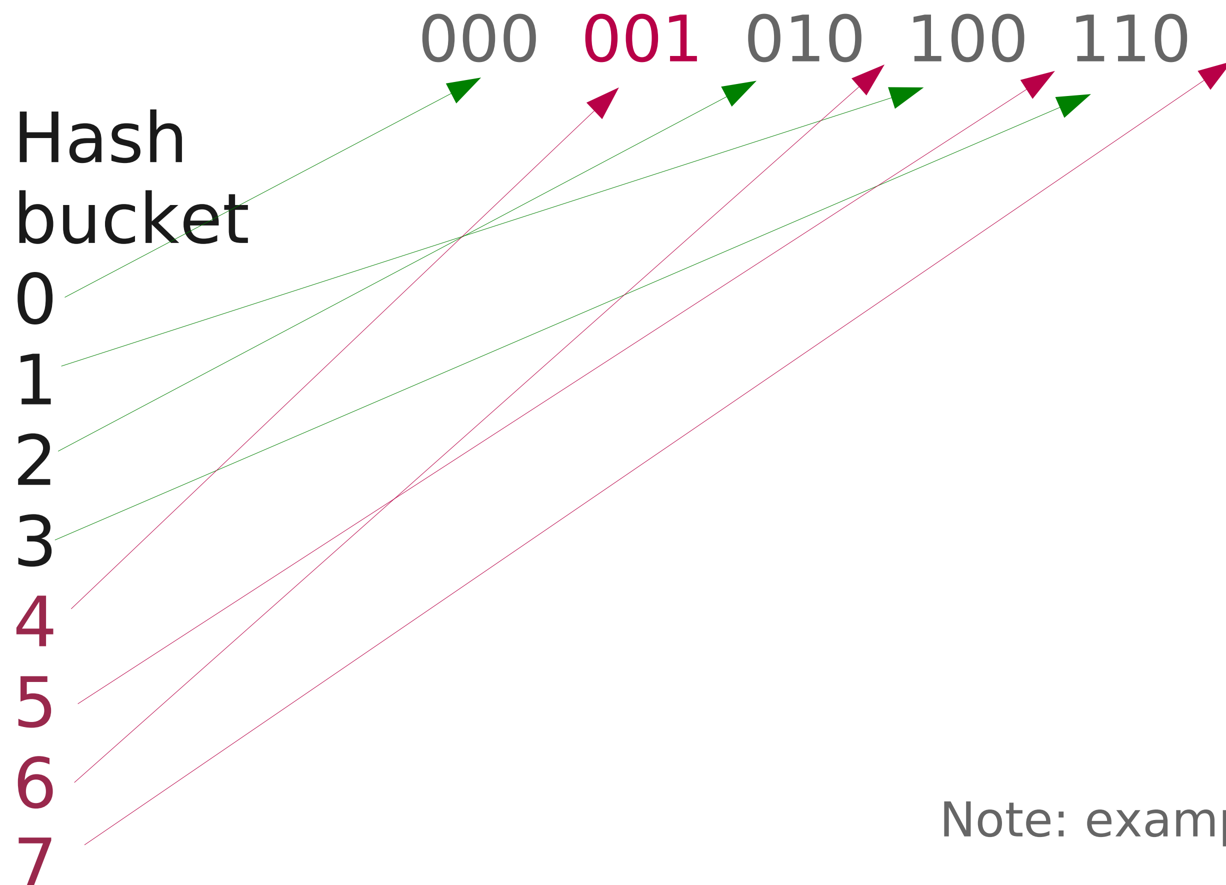
- Based on prior work from
  - Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. Journal of the ACM 53 (May 2006), 379–405.
  - Michael, M. M. High performance dynamic lock-free hash tables and list-based sets. In Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures, ACM Press, (2002), 73-82.
- State of the art: Josh Triplett articles.

> RCU Split-Ordered Lock-Free Resizable Hash Table

- git.lttng.org userspace-rcu.git tree dev branches
  - urcu/ht branch (expand only)
  - urcu/ht-shrink (expand and shrink support)

> Split-Ordering (expand)

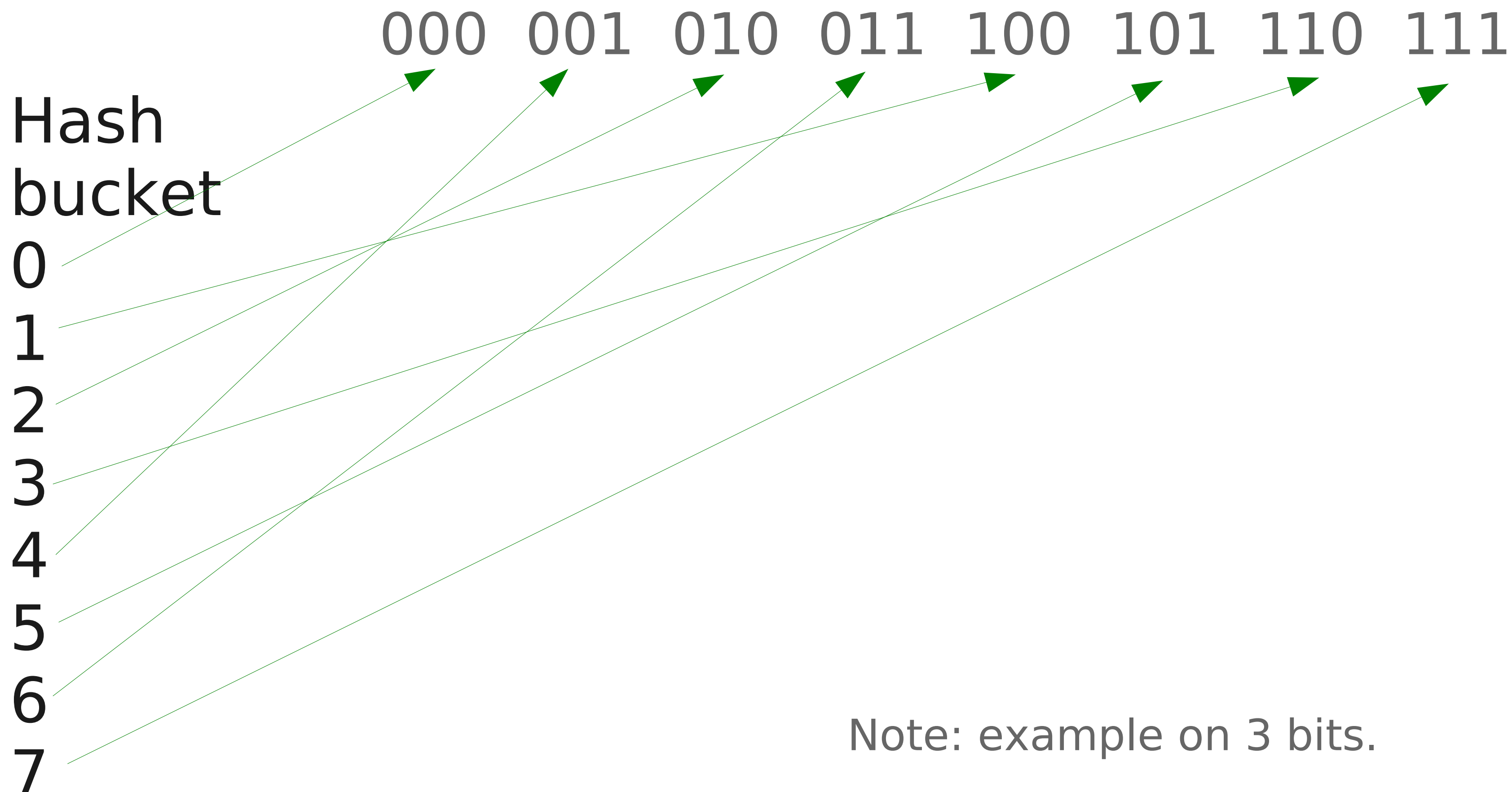
Dummy Nodes (singly-linked list ordered by reversed hash bits)



Note: example on 3 bits.

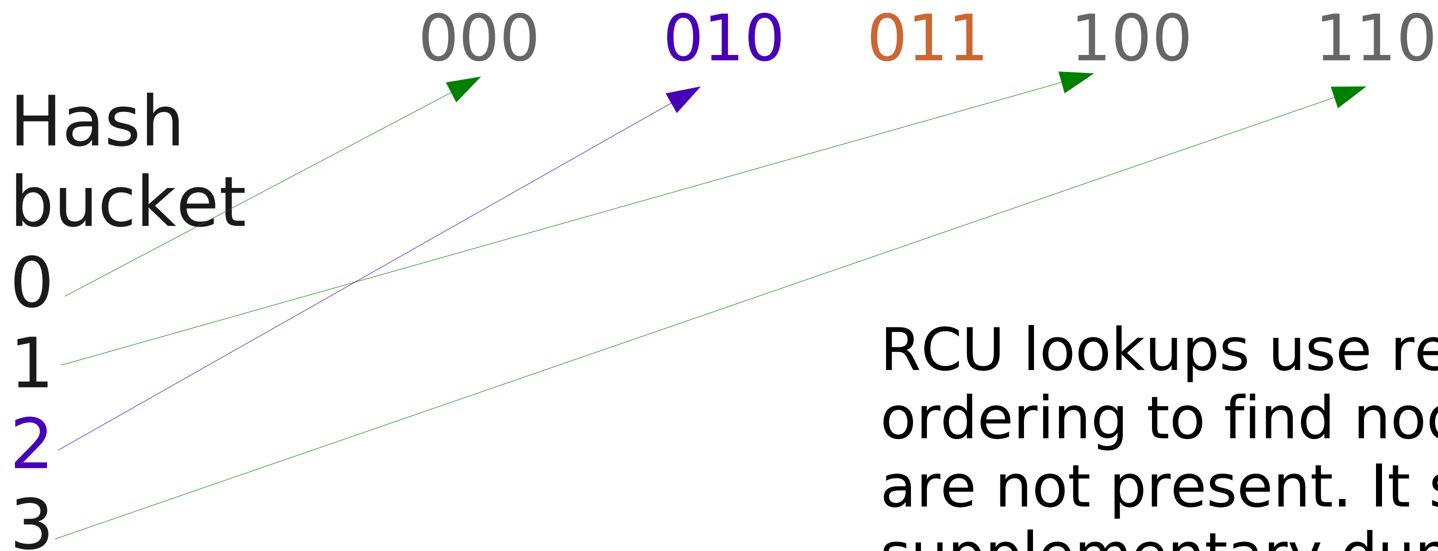
> Split-Ordering

Dummy Nodes (singly-linked list ordered by reversed hash bits)



Note: example on 3 bits.

> RCU Lookups



RCU lookups use reverse hash ordering to find nodes or detect they are not present. It skips over supplementary dummy nodes it encounters, allowing concurrent resizes.

Note: example on 3 bits.

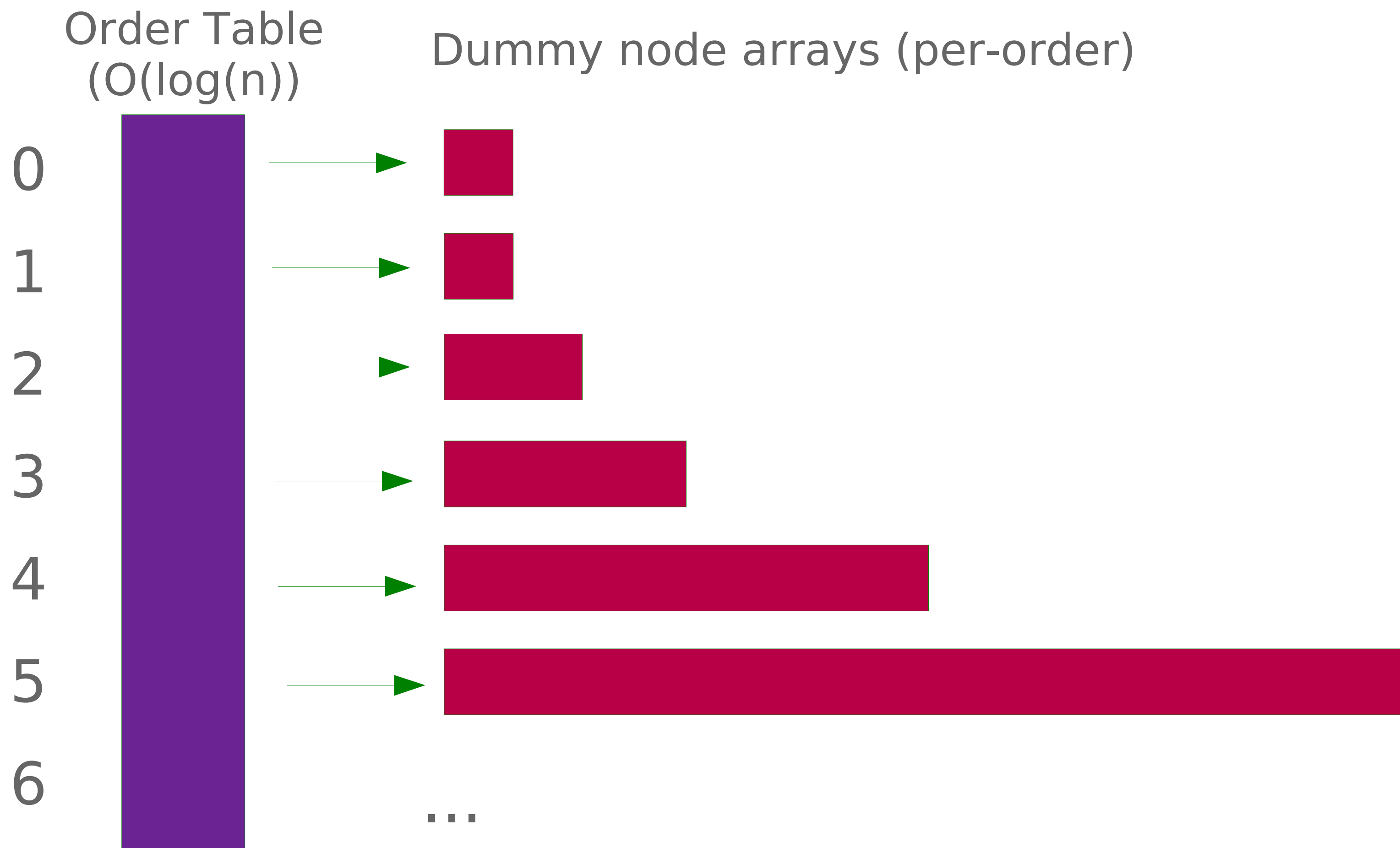
> RCU Hash Table Add/Remove

- Lock-free singly-linked list
  - Logical deletion (removed flag in next pointer) followed by path compression
- Using cmpxchg with RCU read-side lock held to deal with ABA.
- No memory allocated by add/remove.
- add_unique supported.

> RCU Hash Table Resize/Shrink

- Executes concurrently with add/remove/lookup.
- Resize operations are mutually exclusive with each other.
- Re-use add/removal operations to insert dummy nodes.
- Only the top-level lookup table needs to be RCU-aware (lookups skip over extra dummy nodes).
- No node reallocation (in-place resize).

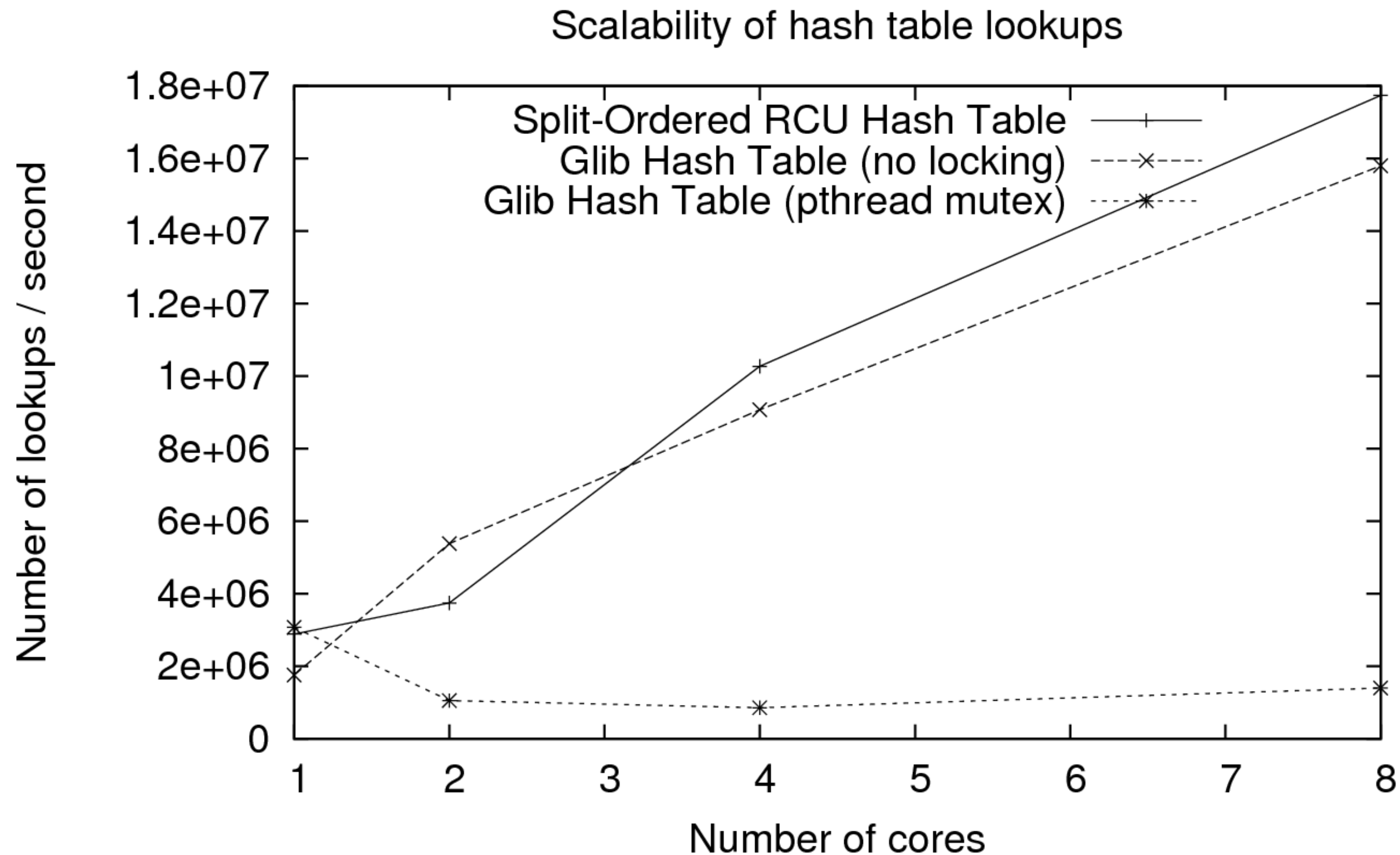
> RCU Hash Table: cache-friendly structure



> RCU Hash Table: automatic resize triggering

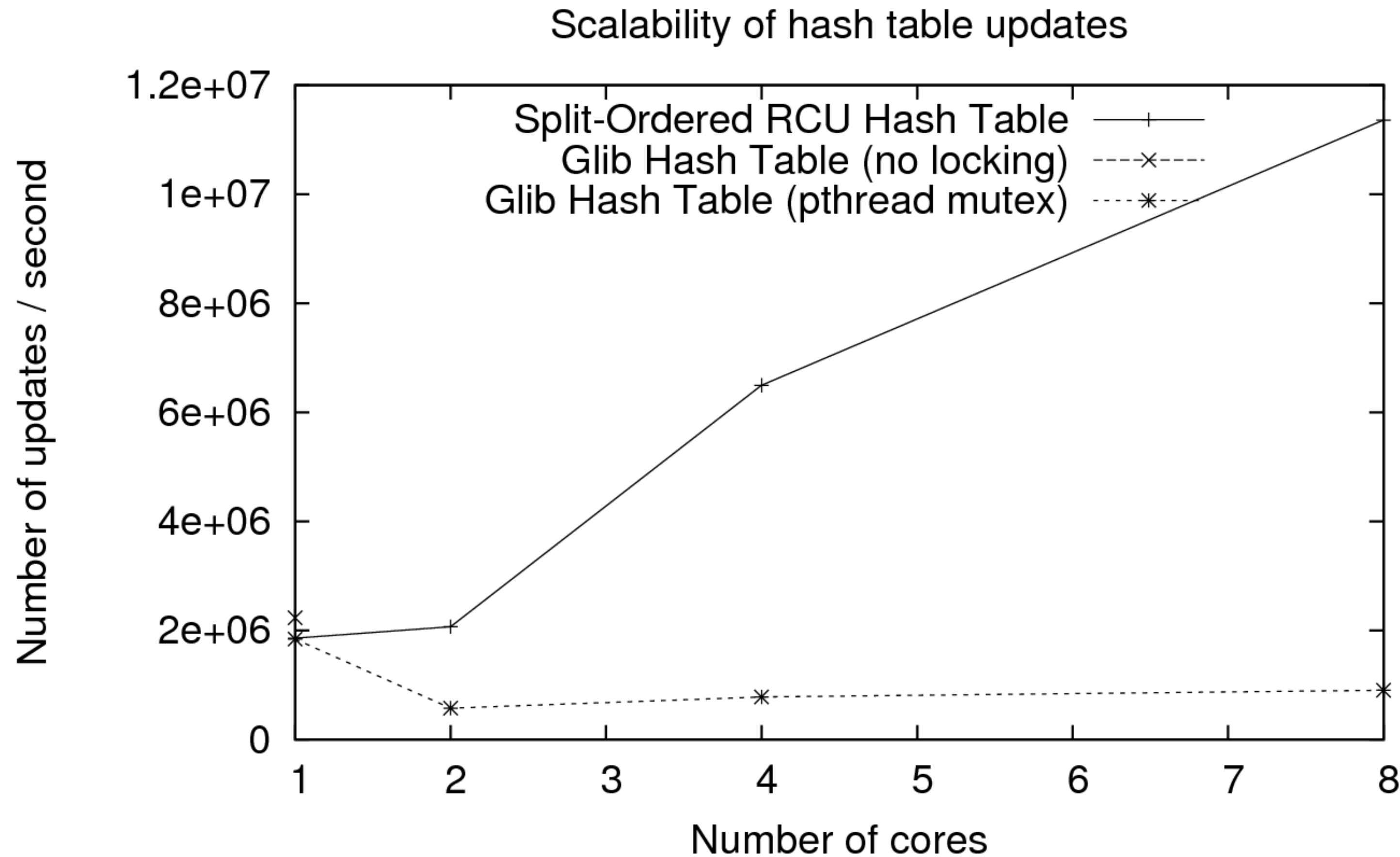
- Table size < 1024 nodes:
  - Expand based on chain lengths (check on node addition). Fine-grained expand-only.
- Table size >= 1024 nodes:
  - Per-CPU split-counters, counting the number of nodes in the table. Coarse-grained expand and shrink.
- TODO: make add/remove help the resize operation (for lock-free guarantee).

> RCU Lock-Free Hash Table (benchmarks)



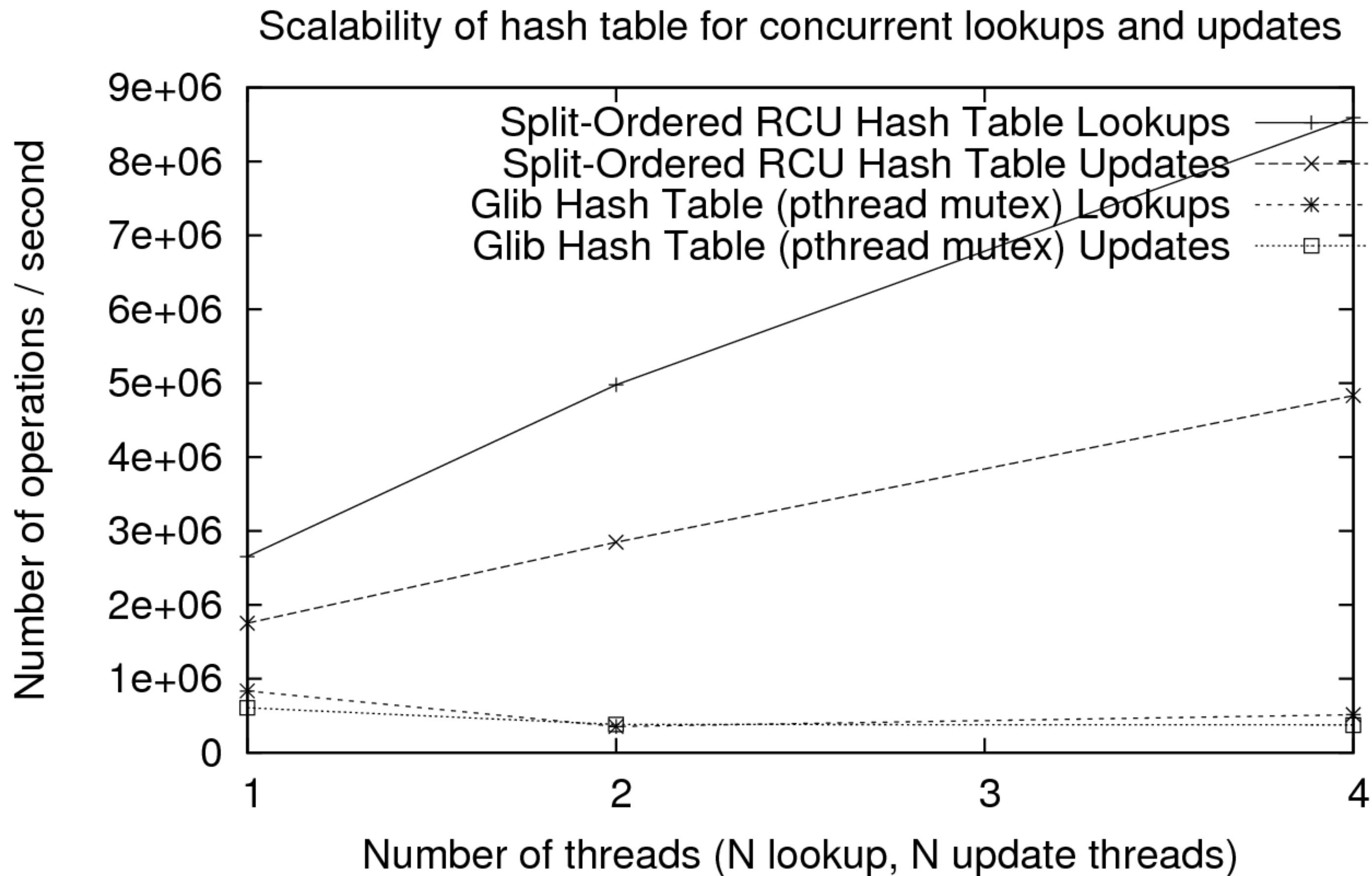
Benchmarks performed on a 2-sockets * 4 core/socket Intel Xeon Core2 2GHz with 16 GB ram.

> RCU Lock-Free Hash Table (benchmarks)



Benchmarks performed on a 2-sockets * 4 core/socket Intel Xeon Core2 2GHz with 16 GB ram.

> RCU Lock-Free Hash Table (benchmarks)



Benchmarks performed on a 2-sockets * 4 core/socket Intel Xeon Core2 2GHz with 16 GB ram.

> RCU Red-Black Tree

- Implementation of RCU-adapted data structures and operations.
  - based on the RB tree algorithms found in chapter 12 of Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Third Edition. The MIT Press, September 2009.
- State of the Art: Phil Howard articles.
- git.lttng.org userspace-rcu.git tree, rbtree2 branch.

> RCU Red-Black Tree

- RCU-specific adaptation
  - Cluster scheme *.
  - Node generations * (decay scheme *).
  - RCU wait-free lookups and traversals.
  - Updates protected by mutual exclusion, do not need to wait for quiescent state.
  - Tree lookup in O(log(n)), traversal in O(n).
  - Allows duplicated entry values.
  - Range-augmented (not detailed here).

* AFAIK, I made up these terms.

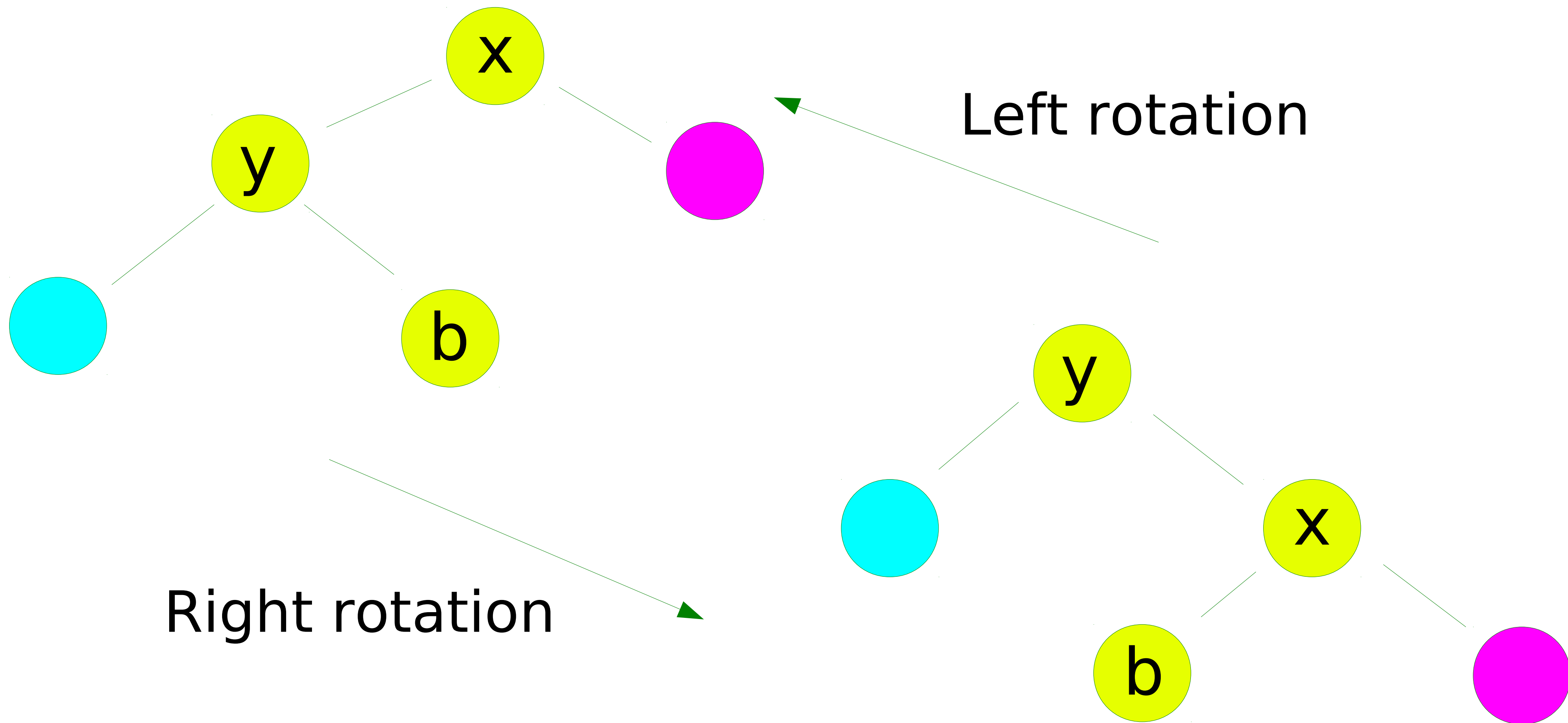
> Cluster Scheme

- A cluster is made of a group of RCU objects that, if taken together as a black box from an external observer point of view, will appear to be unchanged before and after a structure update operation.
- Cluster update overview:
  - Copy cluster, modify cluster copy, set internal pointers, set external pointers to the cluster.

> Cluster Scheme Applied to Red-Black Tree

- Decompose insert/removal into their constituent phases:
  - Rotation : cluster made of 3 nodes. Taken as a black box, the cluster is viewed by observers as the same entity before/after rotation.
  - “Near Transplant”: child takes place of parent. Cluster made of 1 node.
  - “Far transplant” (which I call “Teleport”): a non-immediate child replaces an uppermost parent. Cluster is the entire chain involved between the parent and child (includes child).

> Cluster for Rotations



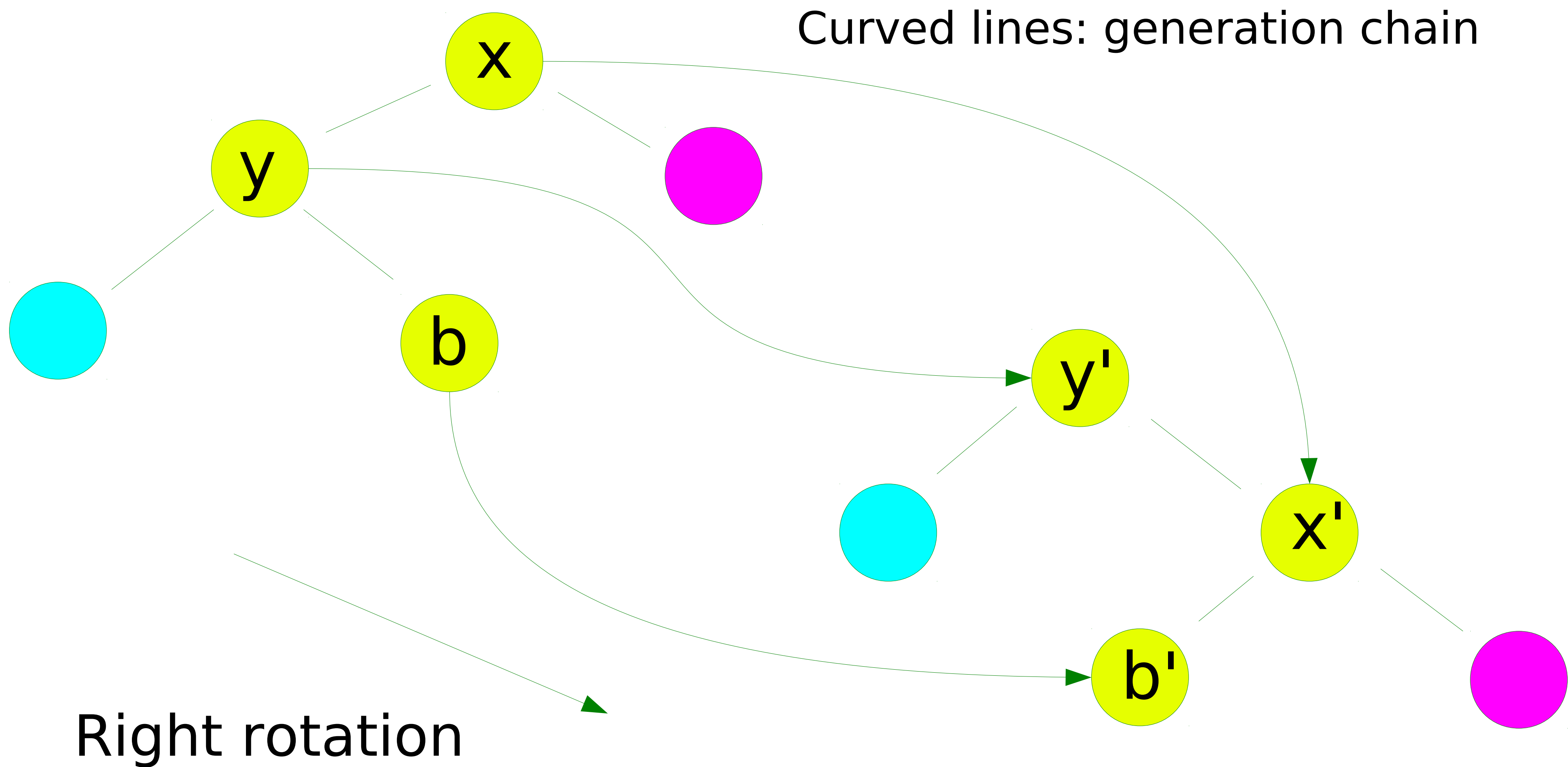
> Node Generations

- Each Red-Black tree operation (insertion/removal) require multiple basic steps (rotations/transplant).
- Balanced Red-Black Tree Algorithm relatively complex (changing its behavior is non-trivial).
- Need scheme that allows to always update the most recent cluster created (no changes lost).

> Node Generations

- Solution: add a linked list of node “generations” in each node.
- Each time a node is duplicated and pending for removal (thus considered “old”), its generation chain pointer is set to the new node version.
- Each time a node is accessed by the algorithms, its generation chain is followed until we reach the most recent node.

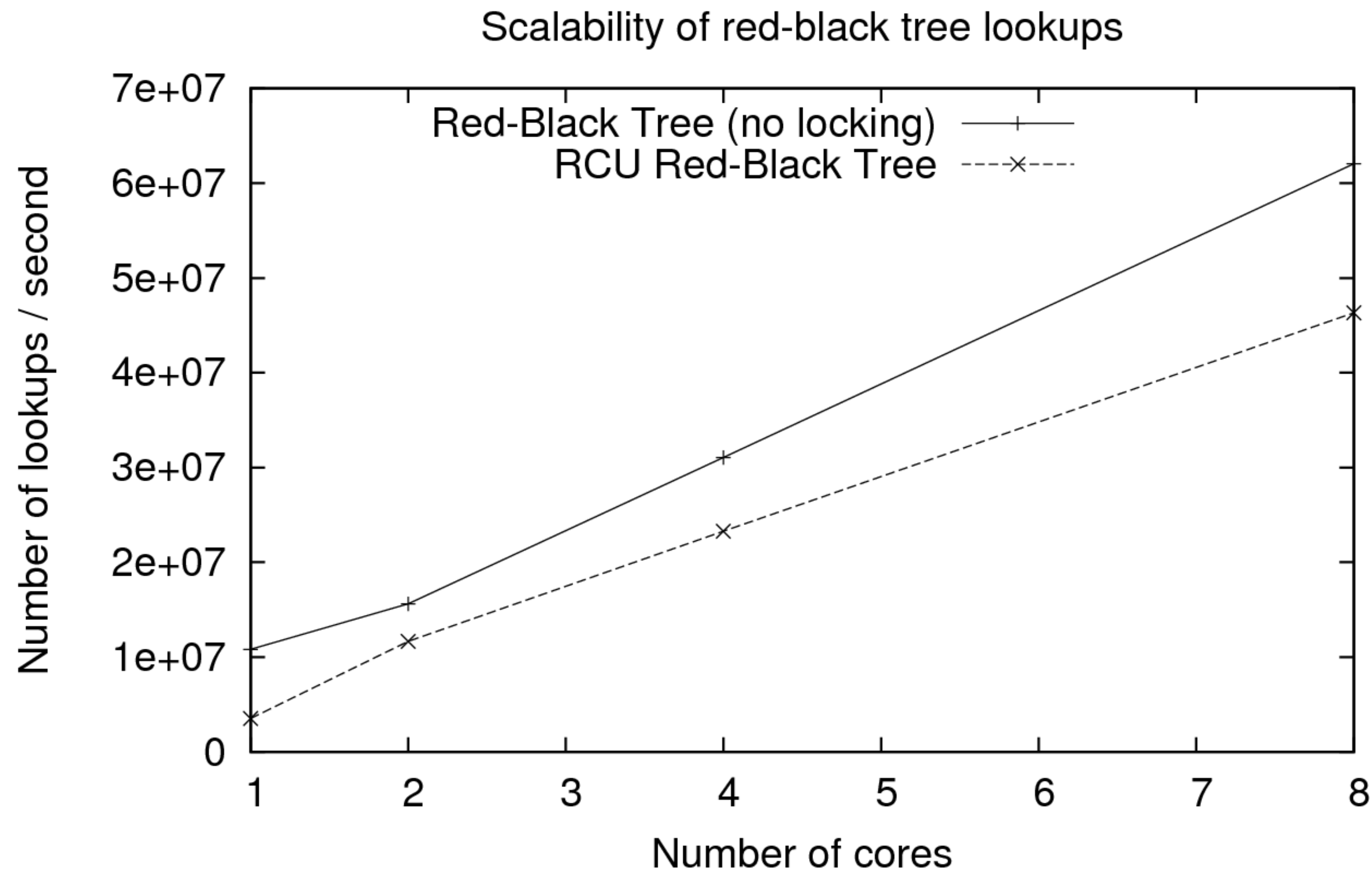
> Node Generations (in 3D!)



> Performance overhead

- Tradeoff: keeping the original algorithm at the expense of frequent memory allocation/call_rcu for memory reclaim.
- We therefore assume the memory allocator and call_rcu are fast enough to provide acceptable update performance.

> RCU Red-Black Tree (benchmarks)



Benchmarks performed on a 2-sockets * 4 core/socket Intel Xeon Core2 2GHz with 16 GB ram.

> RCU Red-Black Tree (benchmarks)

RCU vs non-RCU Red-Black tree comparison
1 writer only (mutex taken for 20 update batches)

	updates/s
RCU	378504
Non-RCU, with mutex	937072
Speedup (RCU : non-RCU)	1 : 2.47

RCU vs non-RCU Red-Black tree comparison
7 readers/1 writer (mutex taken for 20 update batches)

	lookups/s	updates/s
RCU	30931000	378504
Non-RCU, with mutex	43000	937072
Speedup (RCU : non-RCU)	719 : 1	1 : 7.3

> Userspace Wake-up Management

- Direct use of sys_futex, with fall-back on sleep/retry scheme if sys_futex is unavailable.
- No mutex involved, but memory ordering *MATTERS*.
- 1 waker to N read-only waiters
- N wakers to 1 waiter

> 1 waker to N read-only waiters

- For root daemon which needs to signal its present to many unprivileged applications.
- e.g. connect – fail – wait on futex value to become “active” in a shared read-only POSIX memory page.
- Daemon first set futex value to “active”, then awakes all waiters on futex.
- Daemon sets futex value to inactive and closes socket when on teardown.

> N wakers to 1 waiter

- Useful for RCU implementations
  - rcu_read_unlock() are wakers
  - synchronize_rcu() is waiter
- When no waiter is waiting, a simple load and test is executed (small performance overhead for wakers).

> N wakers to 1 waiter

- Waker unconditionally wakes the waiter if it needs to be awakened.
- The waiting state is attached to a complex condition, possibly changed from “false” to “true” by the waker (non-atomically). This condition is sampled by the waiter.

> N wakers to 1 waiter

```
int32_t value;

void waiter(void)
{
  for (;;) {
    value = -1;
    /* Store value before load condition */
    cmm_smp_mb();
    if (cond_is_true()) {
      value = 0;
      break;
    } else {
      if (value == -1) {
         futex(&value, FUTEX_WAIT, -1,
                    NULL, NULL, 0);
      }
    }
  }
}
```

```
void waker(void)
{
  set_cond_true();
  /* Store condition before load value */
  cmm_smp_mb();
  if (value == -1) {
    value = 0;
    futex(&value, FUTEX_WAKE, 1, NULL, NULL, 0);
  }
}
```

> Questions ?

EfficiOS

- http://www.efficios.com

- Userspace RCU Information
  - http://lttng.org/urcu
  - ltt-dev@lists.casi.polymtl.ca