Linux Plumbers Conference Scaling Microconference

RCU Judy Arrays: cache-efficient, compact, fast and scalable trie

E-mail:
mathieu.desnoyers@efficios.com

> Presenter

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

> Content

- Goals of Userspace RCU
- Userspace RCU History
- RCU Lock-Free Resizable Hash Tables
- Judy Arrays
  - vs Red Black trees,
  - RCU-awareness,
  - node compaction,
  - ongoing implementation and next steps.

> Goals of Userspace RCU

- High speed,
- RT-aware,
- Scalable
  - synchronization,
  - data structures,
- ... in userspace.

> Goals of Userspace RCU (2)

- Semantic similar to the Linux kernel,
- Useful for
  - prototyping kernel code in user-space,
  - porting kernel code to user-space,
- LGPLv2.1 license,
- Supports various architectures, and POSIX OSes.
- Linux most optimized, with fallbacks for other OS.

> History of Userspace RCU

- Started in February 2009, initial intent to implement RCU in user-space,
- Low-overhead wait-wakeup scheme,
- call_rcu contributed by Paul E. McKenney (June 2011, version 0.6.0), implementing queue with wait-free enqueue.
- RCU lock-free resizable hash tables, presented at LPC2011: merged May 2012, version 0.7.0.
  - Thanks to Lai Jiangshan, Paul E. McKenney and Stephen Hemminger for their help.

> RCU Lock-Free Resizable Hash Tables

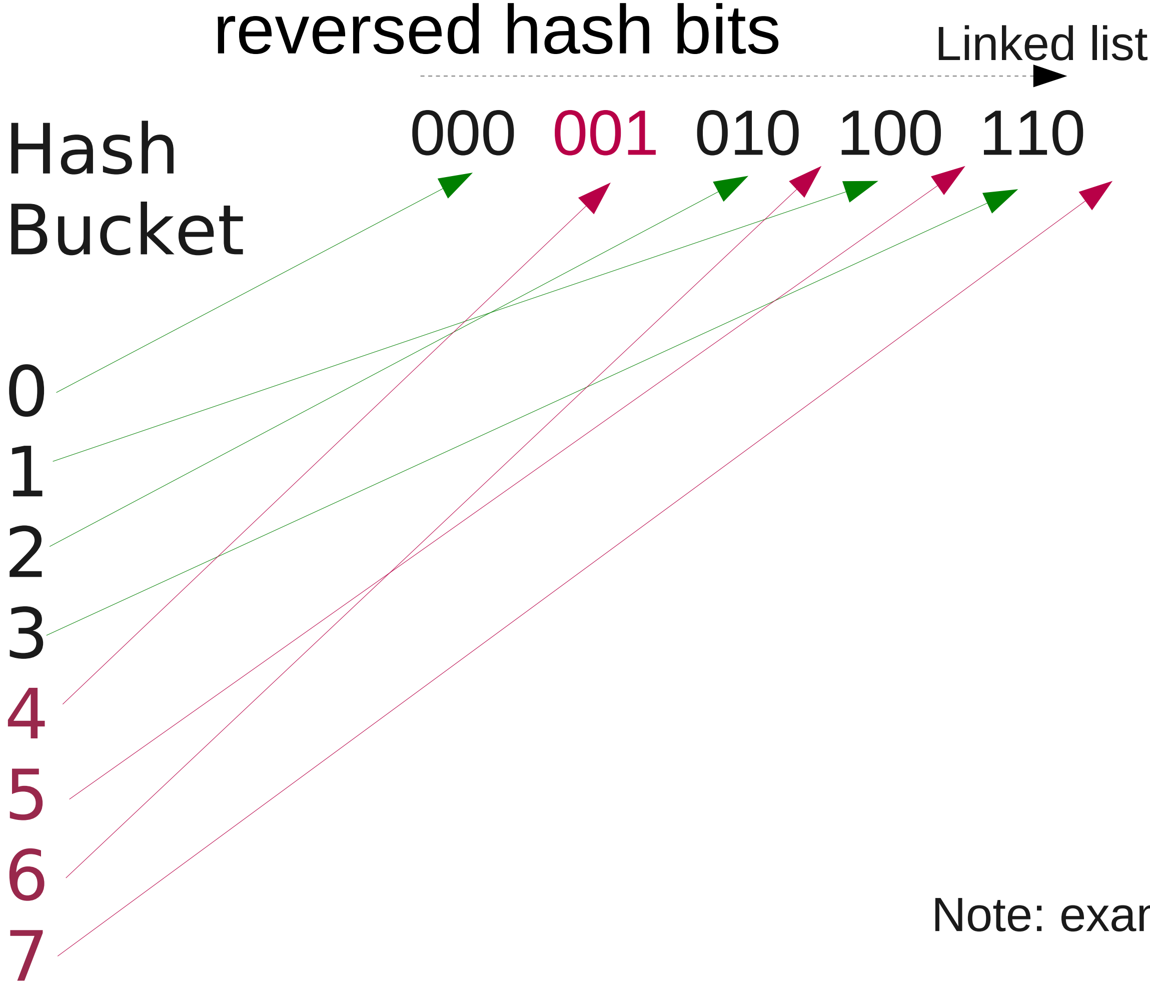
- Wait-free RCU single-node lookup, duplicate traversal, and traversal of the entire table,
- Lock-free updates, supporting:
  - add (with duplicates),
  - add_unique (return previous node if adding a duplicate),
  - add_replace (replace duplicate)
- Updates offer uniqueness guarantees with respect to lookup and traversal operations.

> RCU Lock-Free Resizable Hash Tables (2)

- Hash functions and compare functions are provided by the user,
- Organized as a linked list of nodes, with an index containing "bucket" elements linked within the list,
- On-the-fly resizing, with concurrent lookup, traversal, add and remove operations, is enabled by split-ordering the linked-list (ordering by reversed key bits).

> Split-Ordering (expand)

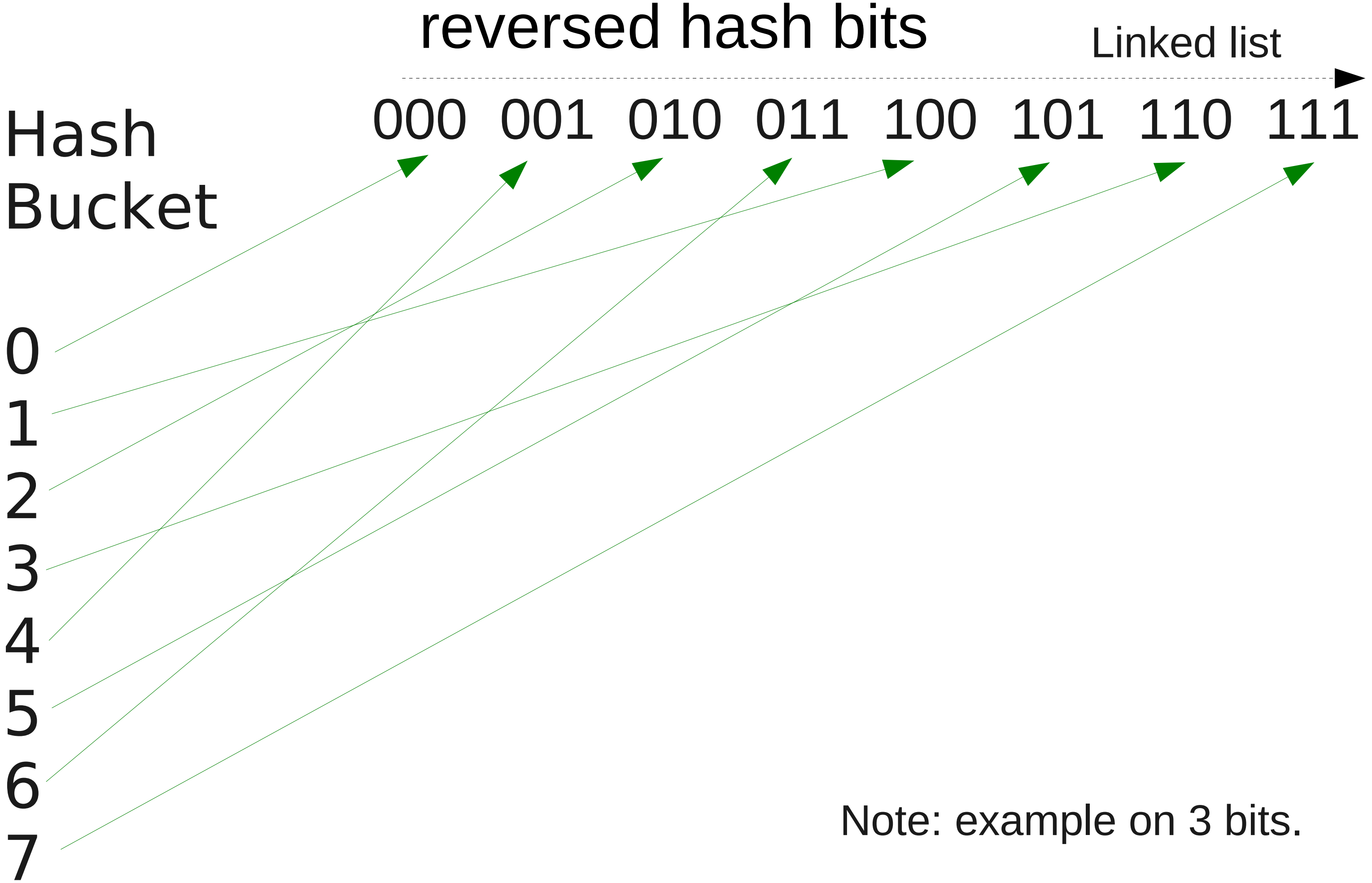
Dummy Nodes: singly-linked list ordered by reversed hash bits



Note: example on 3 bits.

> Split-Ordering

Dummy Nodes: singly-linked list ordered by reversed hash bits



Note: example on 3 bits.

> RCU Lock-Free Resizable Hash Tables (3)

- Automatic resize is triggered by keeping track of the number of nodes in the hash table using split-counters. For small tables, bucket length is used as a trigger.
- Cache efficient index,
- Configurable node index memory management schemes, palatable for 64-bit (linear mapping), 32-bit (order-based) address spaces, or for use with the Linux kernel page allocator (chunk-based).

> RCU Lock-Free Resizable Hash Tables Missing Features

- Rehashing
  - Could probably take a lazy lock, since rare. (combining RCU read-side lock, a flag, synchronize_rcu, and a mutex).
- A hash table does not perform key-ordered traversals, inherent limitation to that structure. (no get next, get previous key)

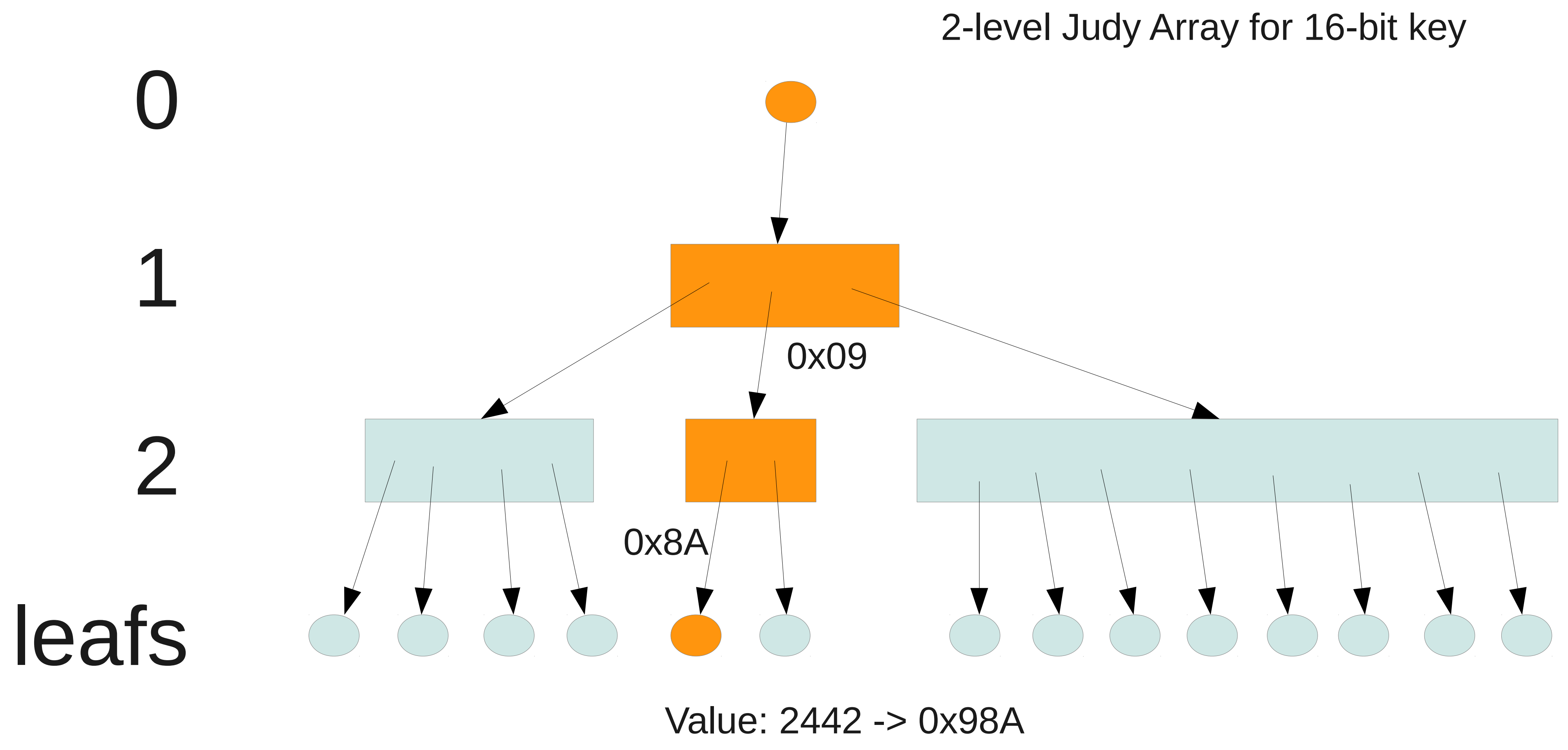
> Judy Arrays

- Jeremy Barnes, from Datacratic, pointed me to this interesting data structure for RCU use,
- Objective: provide a data container that:
  - supports RCU lookups and traversals,
  - allows ordered key traversals,
  - supports scalable updates,
  - cache-efficient,
  - reasonably fast updates.

> What is a Judy Array ?

- An array, indexed by key, for which queries are performed by a lookup through a **multi-level lookup table**. A rule of thumb makes a 256-ary trie a very interesting fit for a level of this lookup table.
- For each 256-ary node, use **node compaction techniques** tailored to the population density of this node to consume less memory.
- Design the node compaction scheme to **minimize the number of cache lines** that need to be accessed per lookup.

> What is a Judy Array ?



> State of the Art of Judy Array

- Invented by HP, LGPL v2.1 implementation
  - http://judy.sourceforge.net/
- Claimed to do better than hash tables,
- Criticized for
  - large and complex implementation (20k LOC)
  - tailored to architecture-specific characteristics
    - cache line size
  - work would have to be re-done as computer architectures evolve.

> Overcomplicated Design ?

- Workshop manual details various special-cases,
- Thought maybe I could find a way to make it relatively simple, yet keeping efficiency, and add RCU-awareness, as well as architecture “future-proofness”.

> Judy Array vs Red Black Trees

- Bounded, smaller number of cache lines touched for lookup in large population:
  - 1M elements, 32-bit key: at most 8 cache lines loaded from memory with Judy (1 or 2 per node), 20 cache lines with RB trees.
- Fixed depth tree based on key size:
  - No rebalancing,
  - No transplant,

RCU-friendly !

- No root node contention when distributing locks across the internal nodes with Judy.

> Judy Array vs Red Black Trees (2)

- No free lunch:
  - need to perform node compaction in Judy,
  - compared to fixed number of tree rotations and transplant in Red Black trees.

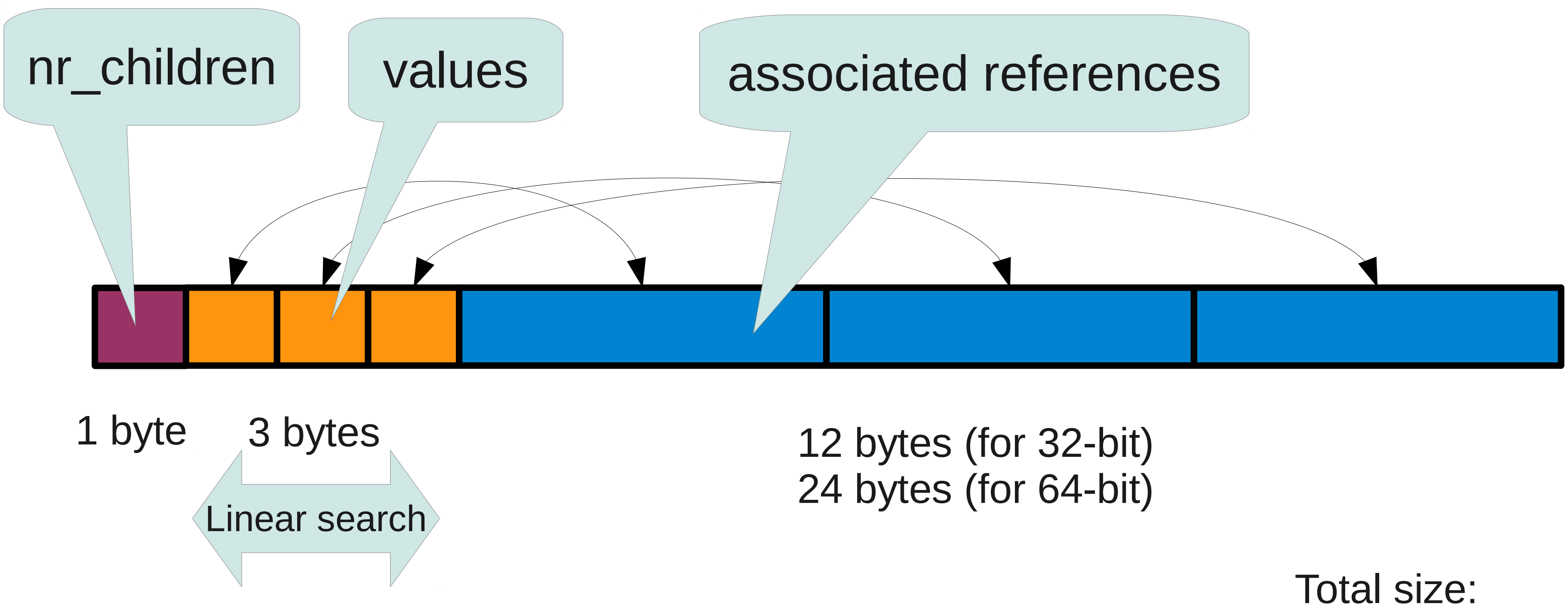
> RCU-aware Node Compaction

- Node reference:
  - Pointer to a node,
  - Low bits contain compaction scheme selector,
  - NULL pointer indicates no child.

> Compaction Scheme: Linear

- Layout
  - 8-bit unsigned integer: number of children populated
  - Array of 8-bit values,
  - Array of references (associated to values).
- 2 cache-line hits per successful lookup
  - 1 for nr_children and array of values,
  - 1 for associated reference.

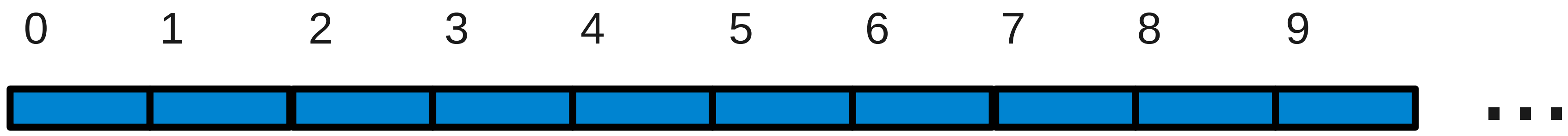
> Compaction Scheme: Linear (2)



Total size:
16 bytes (32-bit)
28 bytes (64-bit)

> Compaction Scheme: Pigeon Hole

- Pigeon Hole array,
- Simple array of 256 references, indexed by value.
- 1 cache line hit per successful lookup.

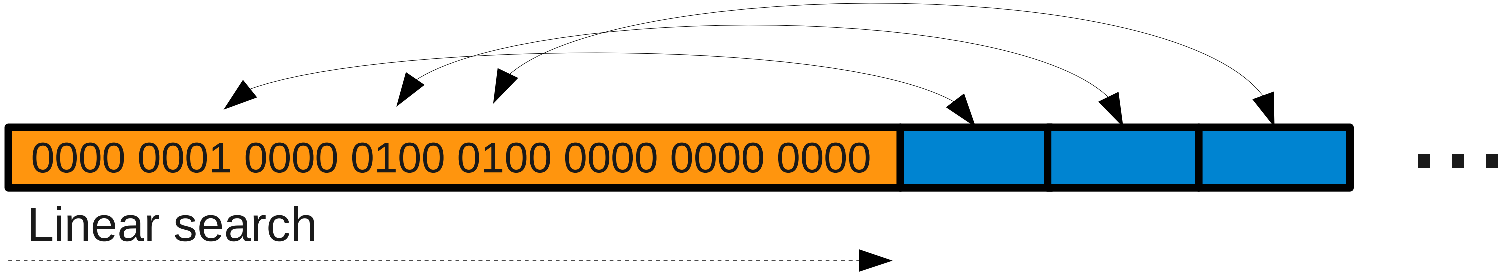


> Portability

- Compaction scheme tailored to each power of two node size,
  - Architecture independency, future-proofness,
- Need 8 compaction schemes that go from 1 to 256 children node compaction schemes.
  - 8 to 1024 bytes on 32-bit,
  - 16 to 2048 bytes on 64-bit.
- A compaction scheme is missing to fill range between 2-cache-line hit “linear” and “pigeon hole” compaction schemes (2 sizes missing).

> Bitmap (HP solution)

- Bitmap of 256-bit (32 bytes), fits in a cache line,
- Count active bits before the one looked up, get associated reference in following array (2 cache lines hit)
- Not RCU-friendly for delete: need reallocation at <u>each</u> delete.
- I thus prefer not going down that route.



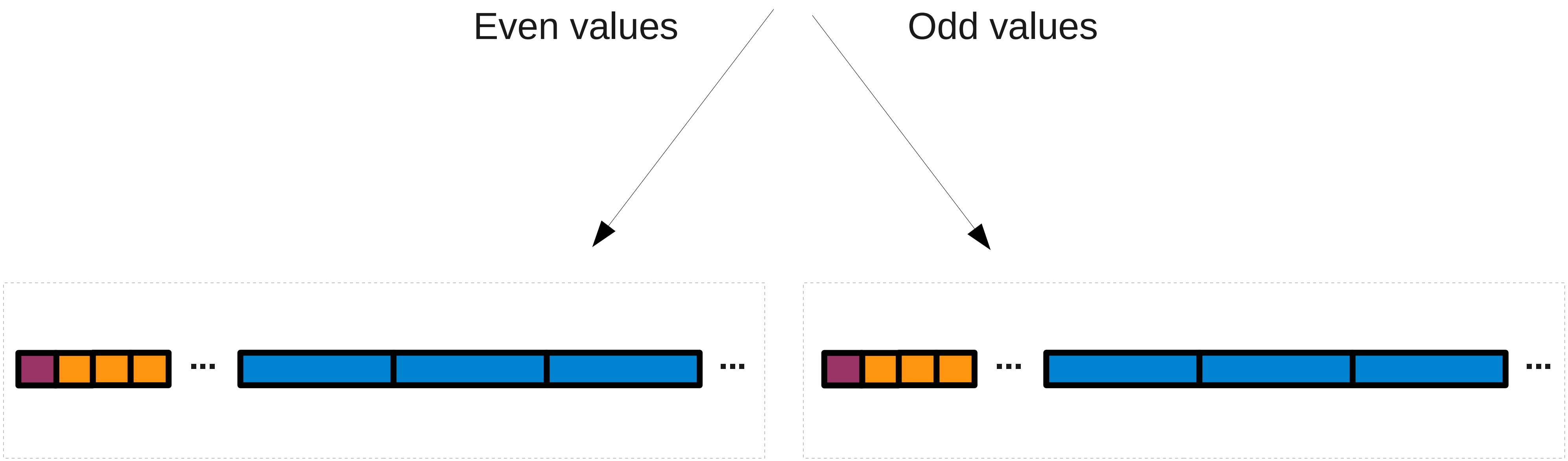
> Pool of Linear Arrays

- Build on the RCU-aware linear array nodes,
- Array of Linear Arrays,
- Split population of a node given a distribution into the respective linear array,
- e.g.: event/odd values could decide the population distribution into one of 2 linear arrays,

> Pool of Linear Arrays (2)

- Even/odd is a choice of bit for distribution,
- Could be any of 8 bits of the keys,
- Choose the best bit choice to minimize unbalance of number of children in each linear array,
- This bit choice can be encoded as part of the encoding scheme selection in reference low bits.
- 2 cache line hits per successful lookup.

> Pool of Linear Arrays (3)



> Pool of Linear Arrays (4)

- Finding the worse possible unbalance for any given key distribution, given we can select the best bit for the given distribution, looks like a NP hard problem (not proven),
- Performed simulations with random distributions to find statistically good limits to trigger recompaction (> 99% of cases),
- Fall back on pigeon hole array if population does not fit.

> Shadow Nodes

- Extra data needed for updates
  - Locks, number of children within pigeon-hole array (to trigger recompaction on removal), rcu head pointer for delayed reclaim,
- Extra augmented range information,
- Locate this information outside of cache lines touched by lookups, outside of power-of-2-sized nodes to limit memory space waste,
- Use RCU lock-free hash table to map nodes to shadow nodes.

> Locking

- Distributed across internal nodes,
- Always taken from the bottom going up,
- Only nodes modified by add/removal need to have their lock taken,
- Good for update-side scalability for updates in different key ranges.

> Update performance

- Only reallocate on recompaction and change of compaction type (even power of two),
  - Amortized reallocation,
- Add an hysteresis in the min/max values that trigger node type change,
- Ensures add/remove cycles on the same key don't trigger frequent recompaction on min/max boundaries.

> Ongoing RCU Judy Array Implementation

- Warning: work in progress !
- git://git.dorsal.polymtl.ca/~compudj/userspace-rcu urcu/rcuja-volatile branch
- What is implemented at this point:
  - Add,
  - Removal,
  - RCU lookups,
  - Duplicate nodes/key.

> RCU Judy Array: Next Steps

- Testing, testing, testing,
- Benchmarks,
- Implement traversals (get next, get previous),
- Implement bit-distribution selection for pool nodes (currently an arbitrary choice),
- Add support for augmented trees (ranges).
- Could be nice to find ways to calculate the pool distribution worse-cases, if possible.

> Questions ?

- Userspace RCU library available at:
  http://lttng.org/urcu

?

  - http://www.efficios.com
- LTTng Information
  - http://lttng.org
  - lttng-dev@lists.lttng.org