Kernel Range Reader/Writer Locking

Linux Plumbers Conference – September 2017. Los Angeles, CA.

Davidlohr Bueso <dave@stgolabs.net>
SUSE Labs.

SUSE
We adapt. You succeed.

Agenda

1. Introduction
2. Semantics
3. Range lock vs rw_semaphore
4. Tree Optimizations
5. What’s left for upstreaming.

Introduction

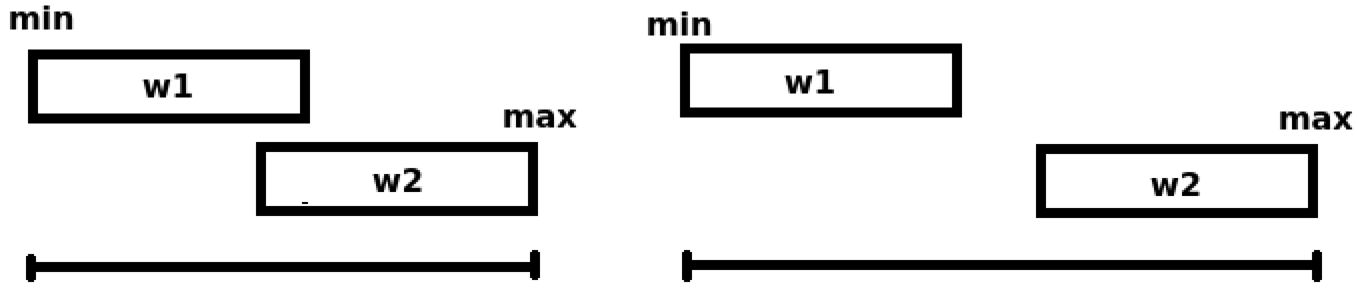
- Why do we want range locking?
  - In ideal scenarios, enables parallelism for non-overlapping ranges.
  - This can be the case for address space (mmap_sem), for example, operating on independent regions.

Introduction

- Why do we want range locking?
  - In ideal scenarios, enables parallelism for non-overlapping ranges.
  - This can be the case for address space (mmap_sem), for example, operating on independent regions.

- With the caveat that the lock isn't *really* a lock.
  - But we call it so because it provides mutual exclusion

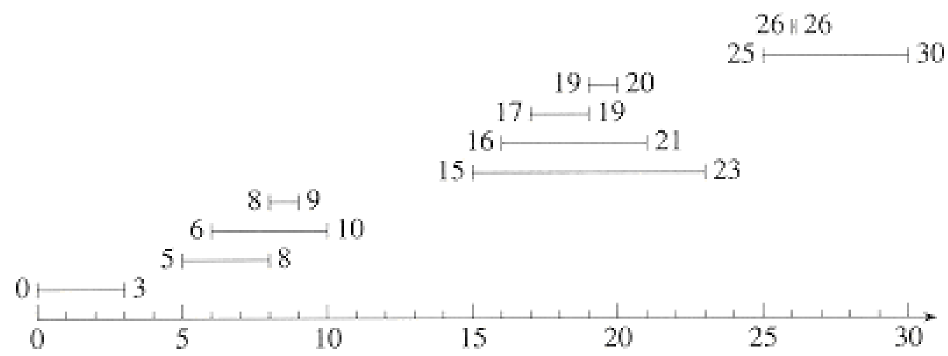
Introduction



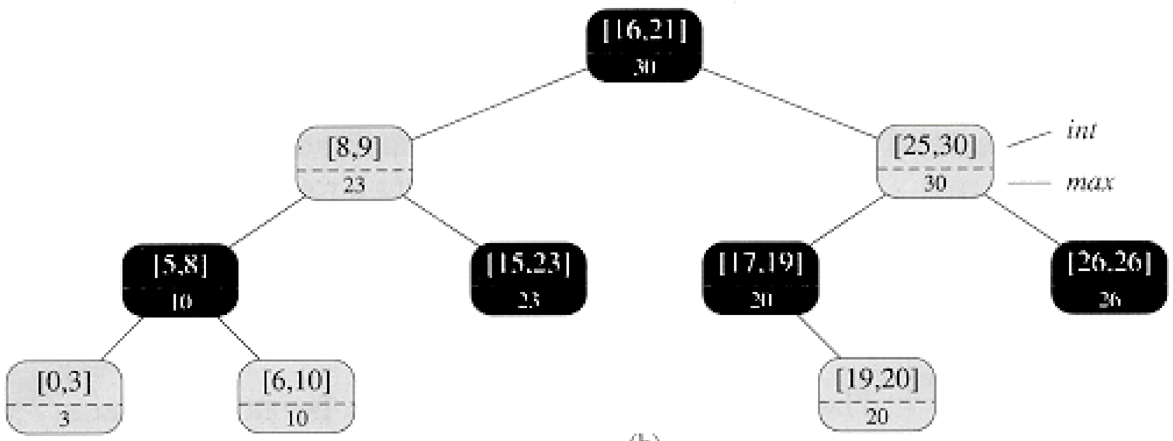
Semantics

- Instead of regular CAS (counter) semantics, range serialization is given by tasks being added to a shared interval tree.
  - Which in turn is an augmented red-black tree.

Semantics



(a)



(b)

Semantics

- Reference counting to account for overlapping ranges.
  - Nodes that overlap without including `current`, whether it be `lock()` or `unlock()`..
  - Task that is adding itself to the tree will block until it's non-zero.

Semantics

- Reference counting to account for overlapping ranges.
  - Provides FIFO ordering. Ie: Assume lock is held by A at `[a,n]` and another thread B comes in at `[g,z]`.

Semantics

- Reference counting to account for overlapping ranges.
  - Provides FIFO ordering. Ie: Assume lock is held by A at `[a,n]` and another thread B comes in at `[g,z]`.
  - Thus ref `[a,n]` = 0 and ref `[g,z]` = 1

Semantics

- Reference counting to account for overlapping ranges.
  - Provides FIFO ordering. Ie: Assume lock is held by A at `[a,n]` and another thread B comes in at `[g,z]`.
  - Thus ref `[a,n]` = 0 and ref `[g,z]` = 1
  - Thread C at `[b,m]` now also tries to acquire the lock (ref = 2)

Semantics

- Reference counting to account for overlapping ranges.
  - Provides FIFO ordering. Ie: Assume lock is held by A at `[a,n]` and another thread B comes in at `[g,z]`.
  - Thus ref `[a,n]` = 0 and ref `[g,z]` = 1
  - Thread C at `[b,m]` now also tries to acquire the lock (ref = 2)
  - Thread A drops the lock, thus ref [g,z] = 0 and ref [b,m] = 1

Semantics

- Reference counting to account for overlapping ranges.
  - Provides FIFO ordering. Ie: Assume lock is held by A at `[a,n]` and another thread B comes in at `[g,z]`.
  - Thus ref `[a,n]` = 0 and ref `[g,z]` = 1
  - Thread C at `[b,m]` now also tries to acquire the lock (ref = 2)
  - Thread A drops the lock, thus ref [g,z] = 0 and ref [b,m] = 1
  - Therefore thread B gets the lock.

Semantics

- Reference counting to account for overlapping ranges.
  - For non overlapping ranges ordering is given by tree traversals (ie two tasks that are awoken).

Semantics

- Reference counting to account for overlapping ranges.
  - For non overlapping ranges ordering is given by tree traversals (ie two tasks that are awoken).

- Starvation wise, there is no lock stealing going and everything is serialized by the `tree→lock`.

Semantics

- Reader/writer.
  - Readers don’t account for other intersecting readers.
  - Tag `task_struct` pointer (LSB) to differentiate.

Semantics

- Requires the caller to setup the ranges before locking it. This is normally local and stack allocated.

- Provides the same calls than regular locks.

```
void range_write_lock(struct range_lock_tree *tree,
                      struct range_lock *lock);

void range_write_unlock(struct range_lock_tree *tree,
                        struct range_lock *lock);
```

Semantics

```
struct range_rwlock myrange;

range_lock_init(&myrange, 10, 100);

range_write_lock(tree, &myrange);
/* do something cool */
range_write_unlock(tree, &myrange);
```

range rwlock vs rw_semphore

- Performance wise a regular lock will always be faster than a range lock. It only helps if it can improve parallelism.
  - Thus this comparison is really a worse case scenario…
  - `range_write_lock_full()`

range rwlock vs rw_semphore

- Range locks have no fastpath.
  - `lock xadd %rdx,(%rax)`
  - Range locking involves at least spin_lock() + spin_unlock() + some loads.

range rwlock vs rw_semphore

- Range locks have no optimistic spinning.
  - Can impact writer threads as they will block.

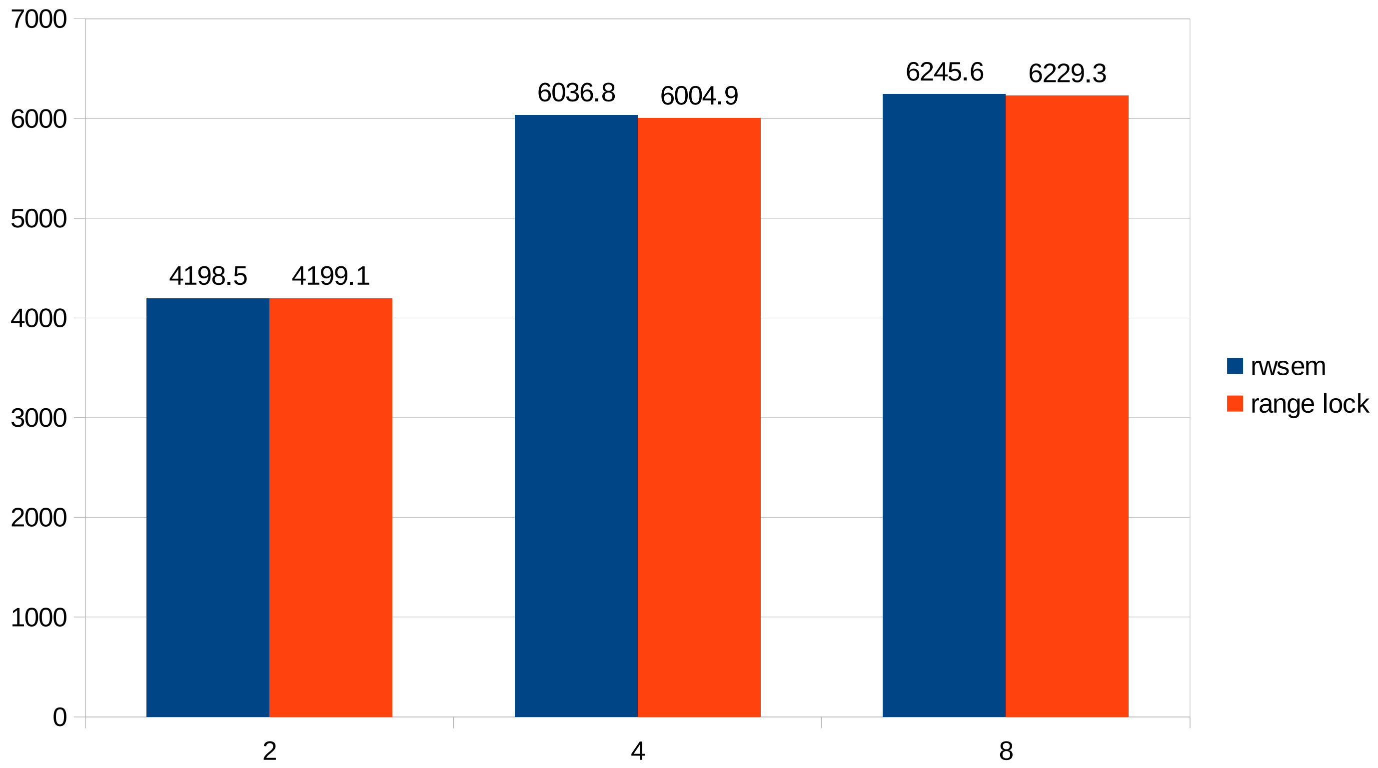
range rwlock vs rw_semphore

- Range locks have no optimistic spinning.
  - Can impact writer threads as they will block.

- Range locks do not favor writers over readers (or vice-versa).

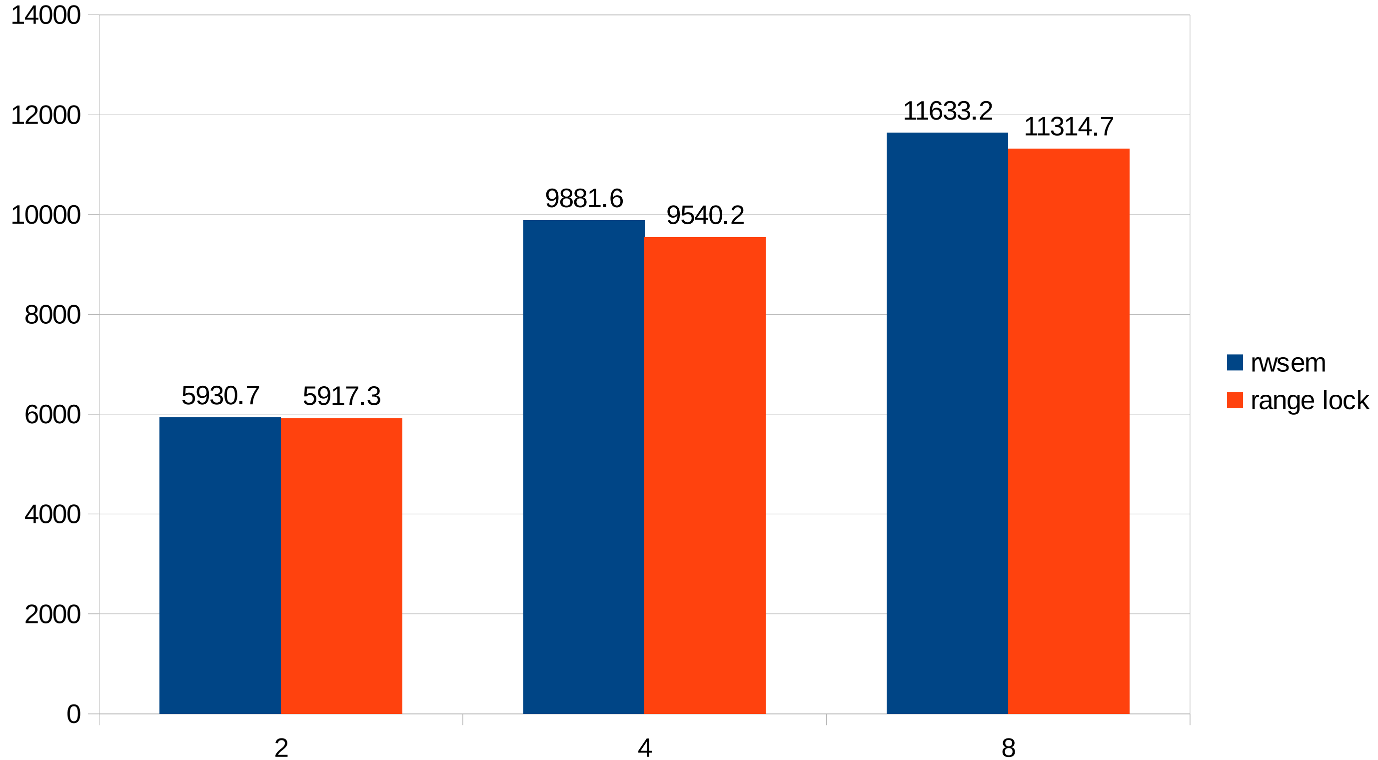
range rwlock vs rw_semphore

- Synthetic 1:1 results (4 core AMD write-only):



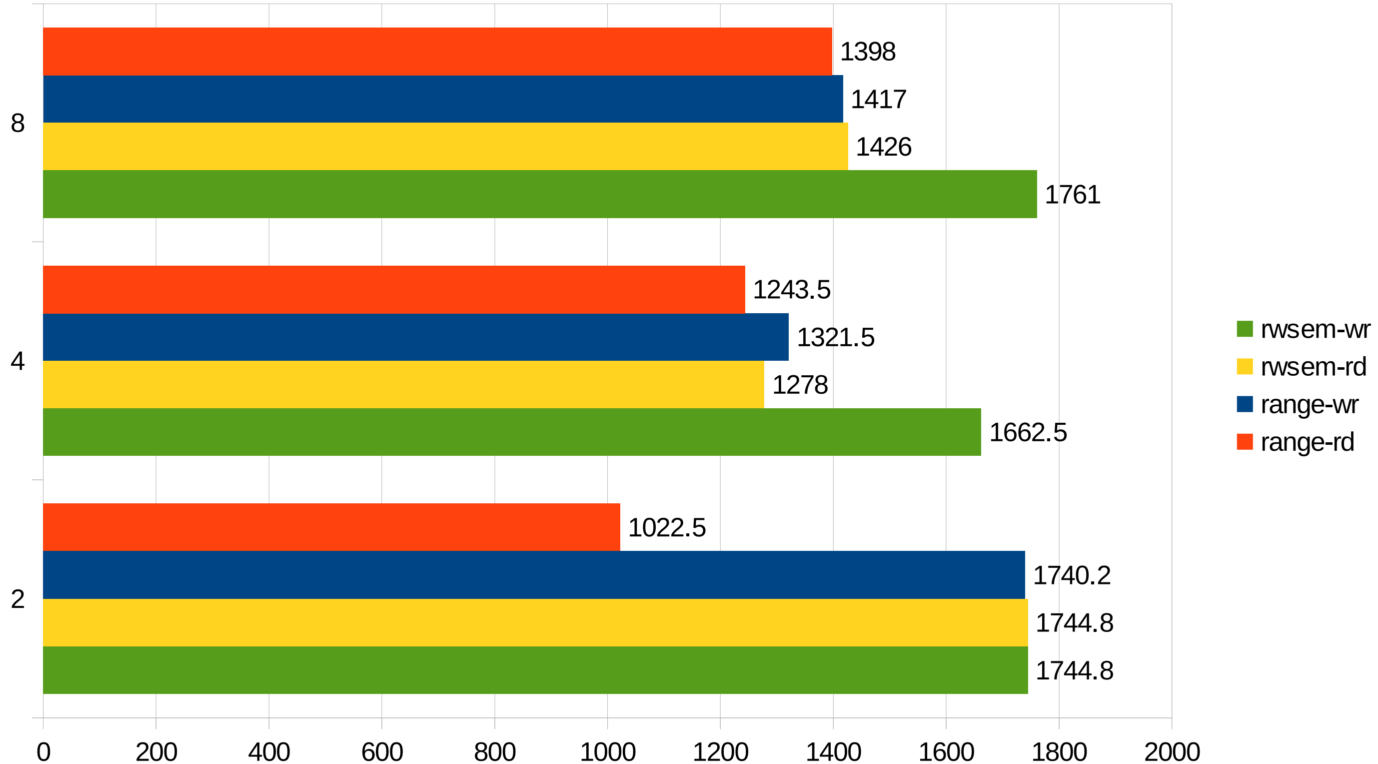
range rwlock vs rw_semphore

- Synthetic 1:1 results (4 core AMD read-only):



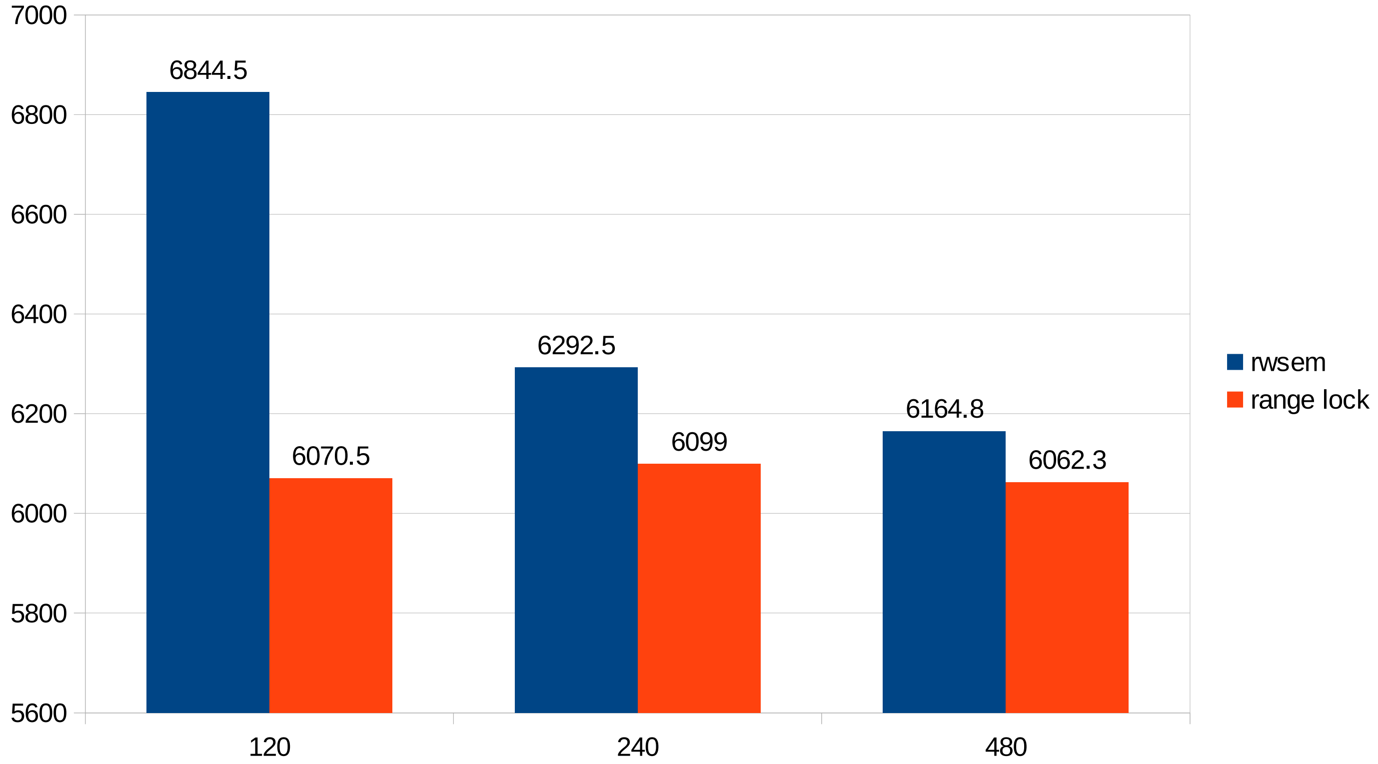
range rwlock vs rw_semphore

- Synthetic 1:1 results (4 core AMD read/write):



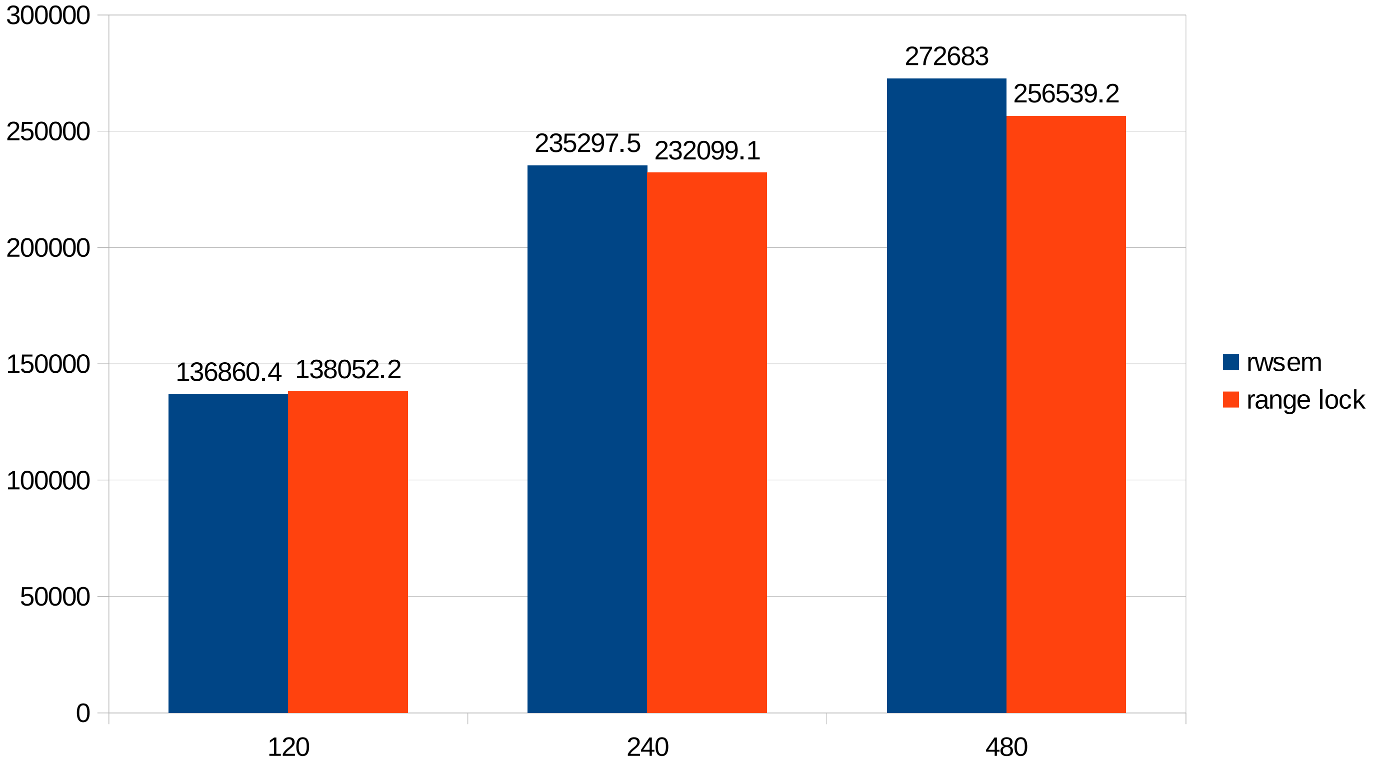
range rwlock vs rw_semphore

- Synthetic 1:1 results (240 core IvyBridge write-only):



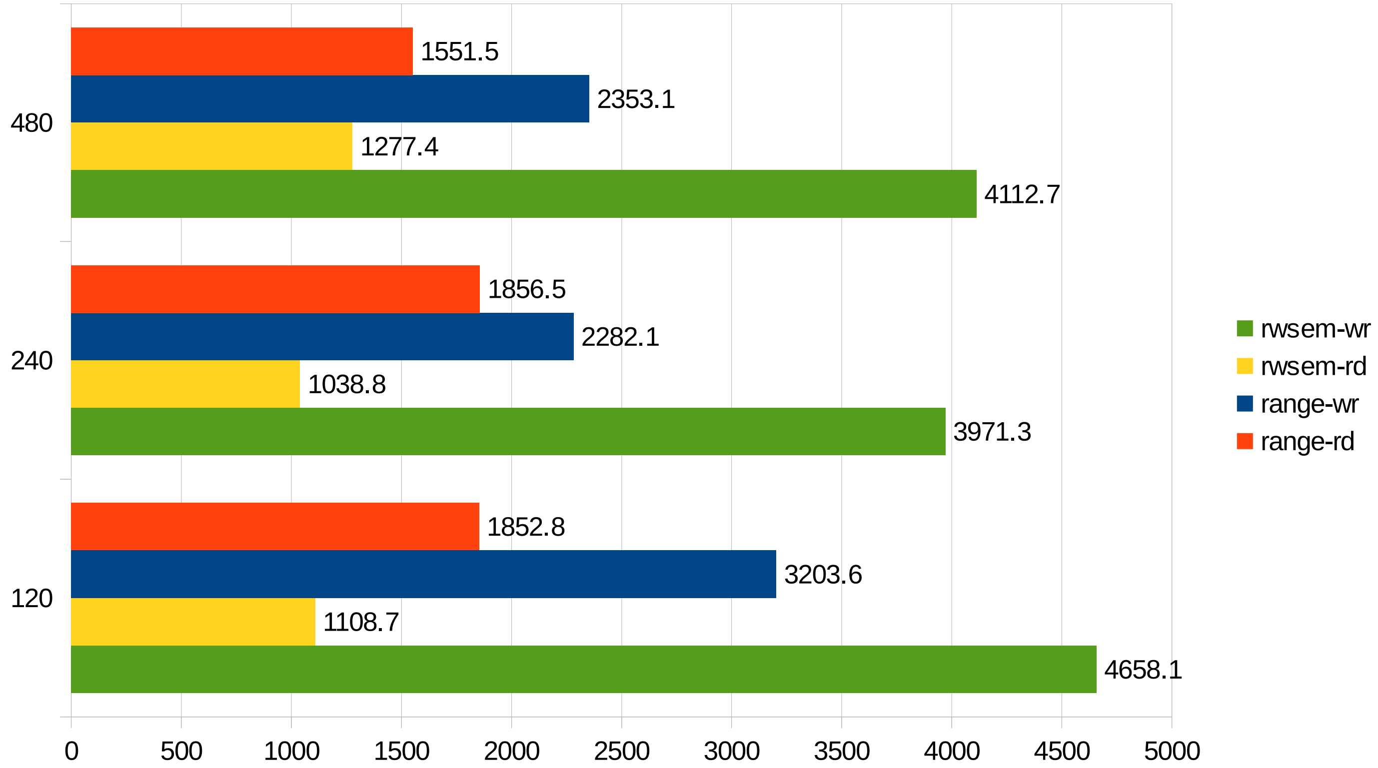
range rwlock vs rw_semphore

- Synthetic 1:1 results (240 core IvyBridge write-only):



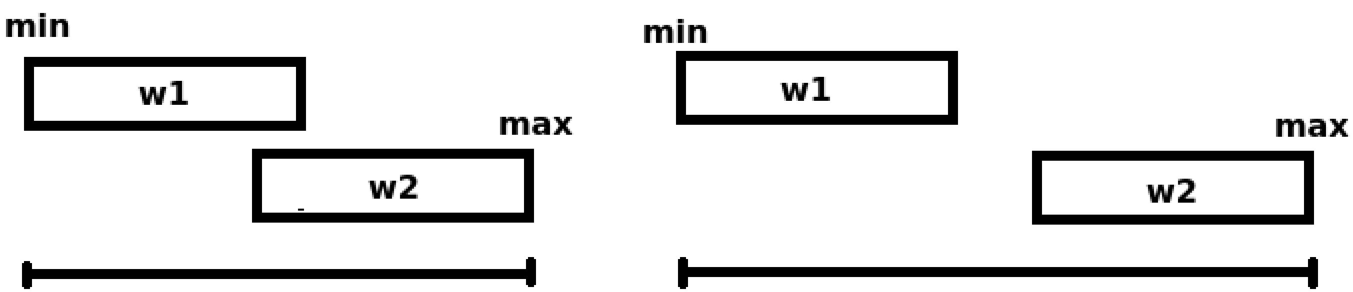
range rwlock vs rw_semphore

- Synthetic 1:1 results (240 core IvyBridge read/write):



Red-Black Tree Optimization #1

- Fast interval tree intersections/overlaps
  - Avoids O(logN) tree walks.



in case of overlap:

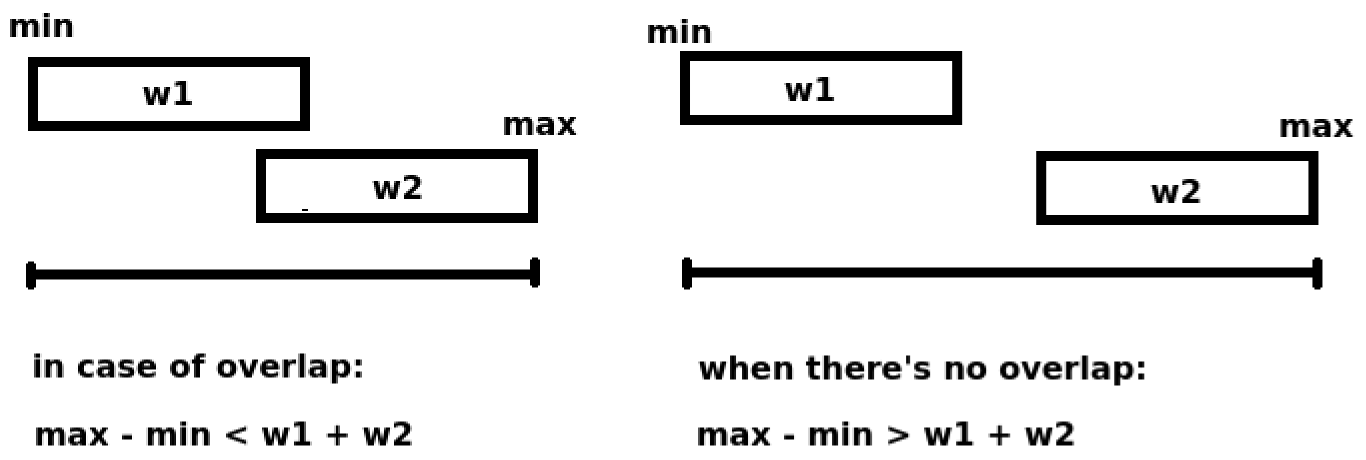
max - min < w1 + w2

when there's no overlap:

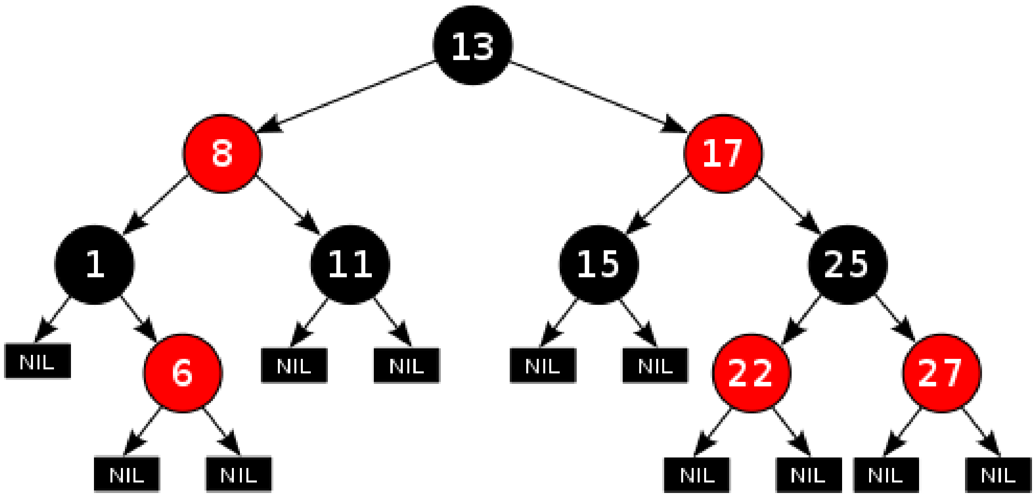
max - min > w1 + w2

Red-Black Tree Optimization #1

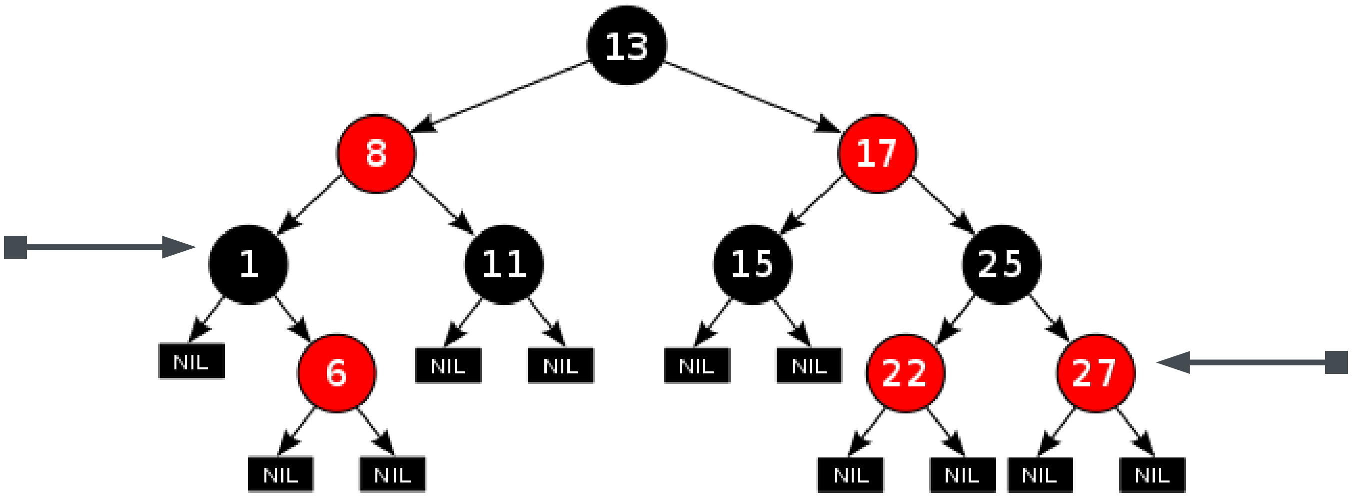
- We need the tree's smallest *start* and largest *end* In O(1).



Red-Black Tree Optimization #1



Red-Black Tree Optimization #1



Red-Black Tree Optimization #1

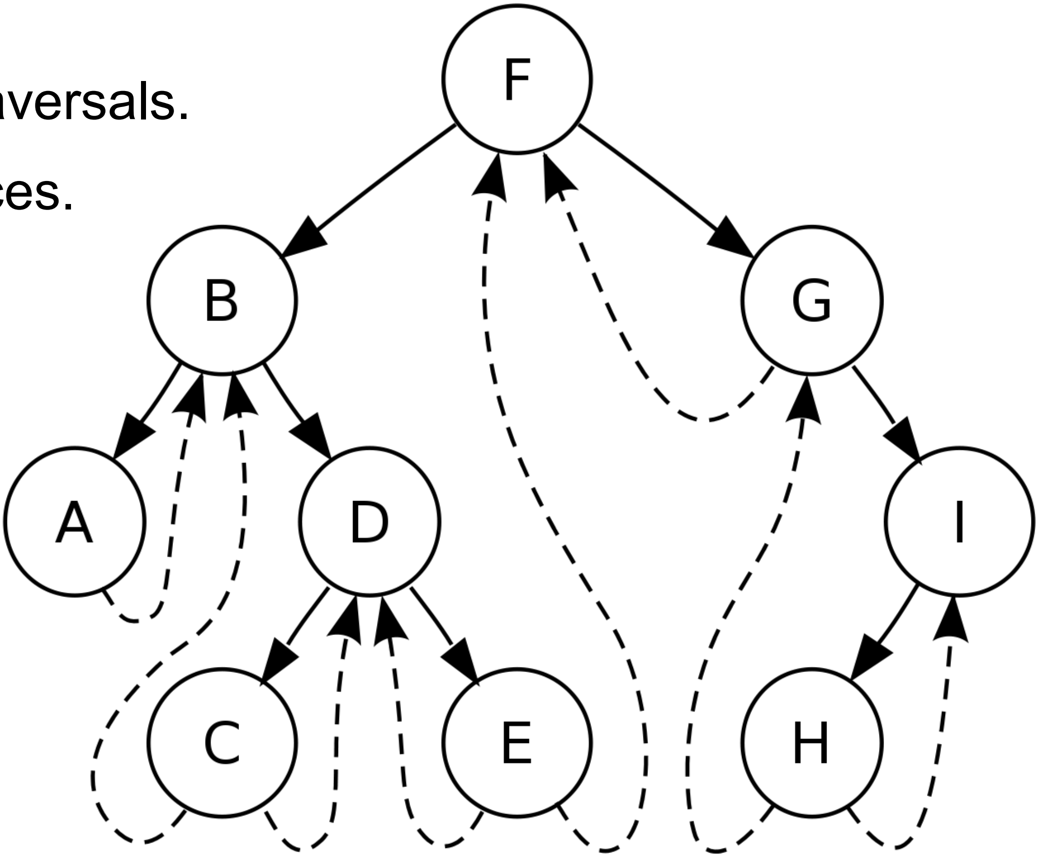
- root’s last-in-subtree for the largest value.

- Cache leftmost node (with the help of the caller, like everything else).
  - `rb_first_cached(cached_root)`
  - `rb_insert_color_cached(node, cached_root, new)`
  - `rb_erase_cached(node, cached_root)`

- In v4.14.

Red-Black Tree Optimization #2

- Threaded rbtrees
  - Allows O(N) inorder traversals.
  - Caveats are rb interfaces.



Red-Black Tree Optimization #2

- Rbtrees have n+1 nil children pointers.
  - These can be reused as threads.
  - Threads are the prev/next inorder node.
  - To not enlarge the data structure, tag the `struct rb_node` pointer (LSB) such that we can tell appart threads and nodes.

Red-Black Tree Optimization #2

```
/* Figure out where to put new node */
while (*new) {
     ...
   parent = *new;
     if (result < 0)
          new = &((*new)->rb_left);
     else if (result > 0)
          new = &((*new)->rb_right);
     else
          return FALSE;
}
/* Add new node and rebalance tree. */
rb_link_node(&data->node, parent, new);
rb_insert_color(&data->node, root);
```

SUSE

Red-Black Tree Optimization #2

```
/* Figure out where to put new node */
while (*new) {
     ...
   parent = *new;
     if (result < 0)
          new = rb_left(*new);
     else if (result > 0)
          new = rb_right(*new);
     else
          return FALSE;
}
/* Add new node and rebalance tree. */
rb_link_node(&data->node, parent, new);
rb_insert_color(&data->node, root);
```

Red-Black Tree Optimization #2

```
/* Figure out where to put new node */
while (*new) {
     ...
   parent = *new;
     if (result < 0)
          new = rb_left(*new);
     else if (result > 0)
          new = rb_right(*new);
     else
          return FALSE;
}
/* Add new node and rebalance tree. */
rb_link_node(&data->node, parent, new);
rb_insert_color(&data->node, root);
```

TODO

- More real world workload testing.
- Get threaded rbtrees upstream.
- Think of ways to avoid `tree->lock` (probably very dangerous).
- Get range locking into the kernel.

Further Reading

- Latest patchset (v3):
  - https://lwn.net/Articles/722741/
- Range reader/writer locks for the kernel (article):
  - https://lwn.net/Articles/724502/

Thank you.

SUSE®

We adapt. You succeed.