



Optimizing igb and ixgbe network driver scaling performance

Alexander Duyck,
LAN Access Division,
Intel Corp.

LINUX PLUMBERS CONFERENCE

SEPTEMBER 7-9, 2011 SANTA ROSA, CALIFORNIA

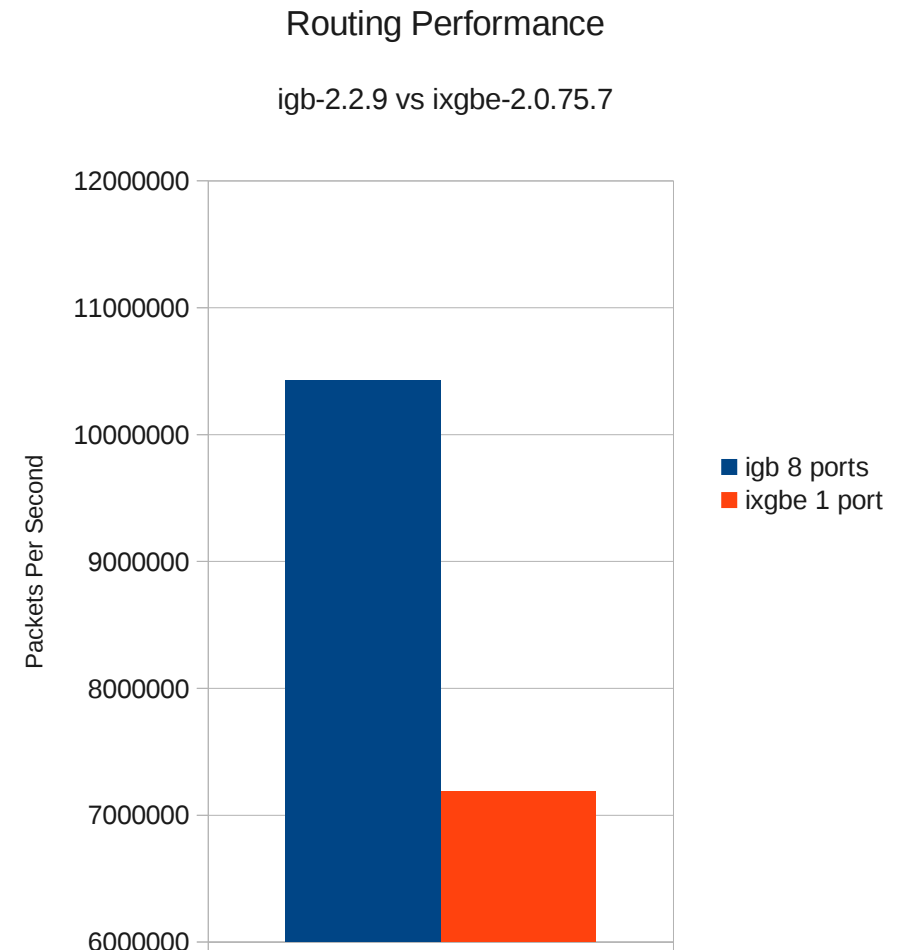
Agenda

- The state of the igb and ixgbe drivers then and now
- Configuring the kernel and system for best performance
- Getting to the root cause of the performance improvements
- Where we still might have room to improve



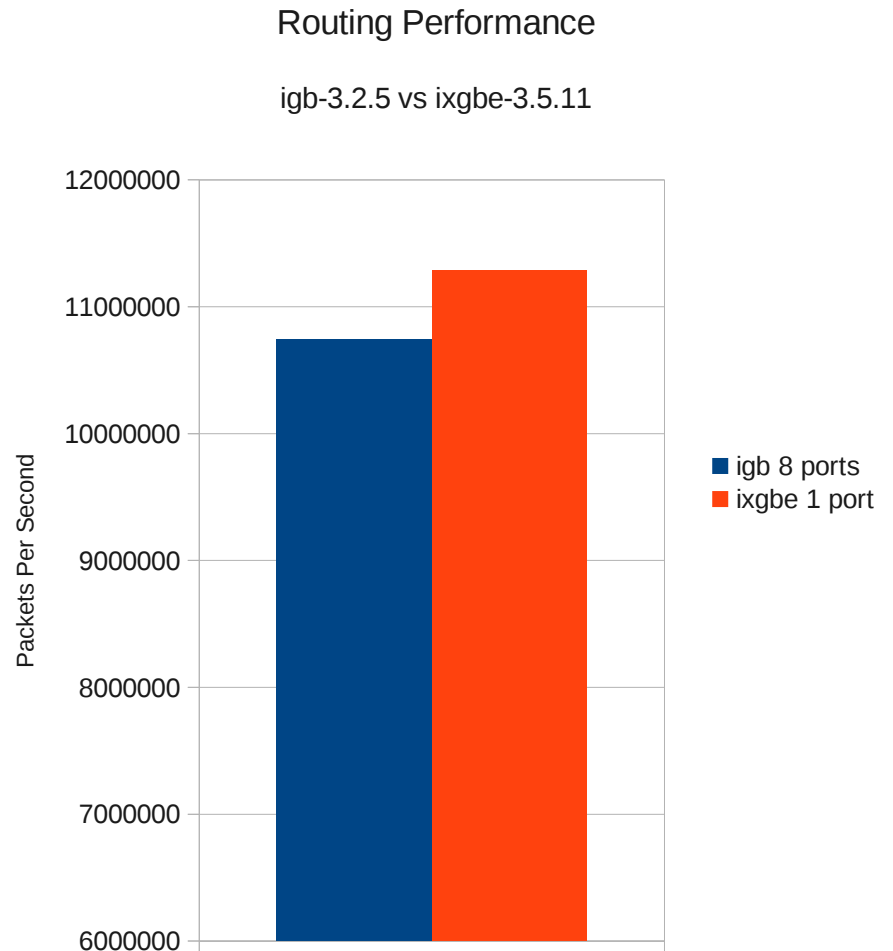
The state of igb & ixgbe over a year ago

- Problem: igb was over 40% faster than the ixgbe.
- Solution: Refactor ixgbe to more closely match igb.



The state of igb & ixgbe now

- Mission accomplished!
- How did we get from there to here?
- Could we be masking over some other issues?
- Where do we go from here?



Configuring the kernel for best performance

- Disable any and all config options that add to the size of the sk_buff w/o any benefit to your testing
 - Ipv6, IPSEC, IOAT, Netfilter, Qos support, and network actions
 - Result should be an sk_buff that fits in 3 cache lines
 - May be unrealistic but we are testing the drivers, not the stack
- Disable IOMMU
 - Generates significant DMA map/unmap overhead
 - May also be disabled via kernel parameter



Configuring the system for best performance

- Evenly load the nodes and memory channels with memory
 - The test system had 2 nodes with 3 channels of memory each.
 - I loaded 1 2G DIMM of memory on each channel for a total of 12GB
- Evenly distribute device interrupts on CPUs
 - `set_irq_affinity.sh` script included with `ixgbe` driver can now handle this task



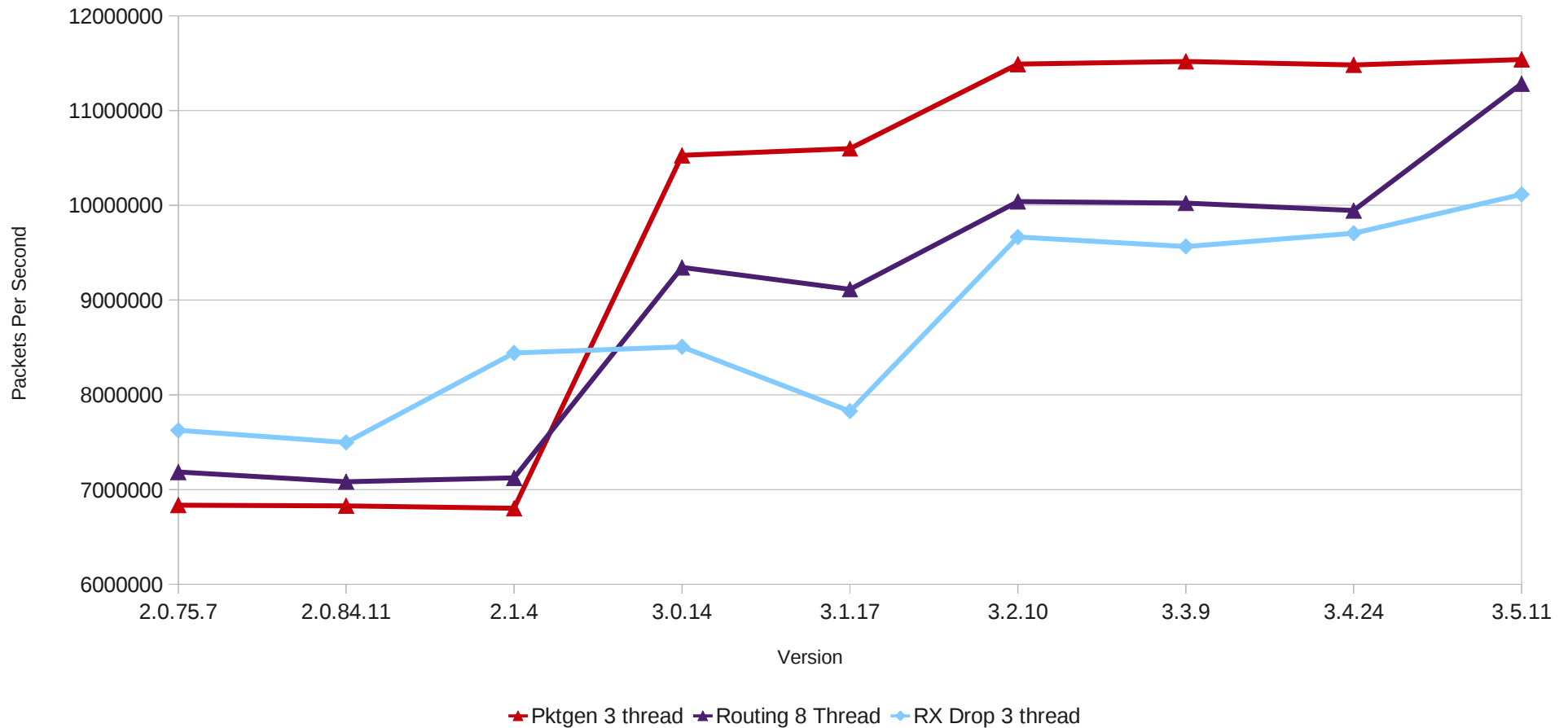
The test configuration

- System running dual Xeon X5680 @ 3.33Ghz
 - Running 2.6.35.14 kernel
 - Connected back to back with Spirent Smartbits 6000c containing a XLW-3720a card
 - Tests typically ran with 64 simultaneous UDP streams
- 3 Basic tests
 - 3 Queue pktgen
 - 3 Queue receive & drop at ip_rcv
 - 8 Queue bidirectional single port routing
- Why select only 3/8 queues?
 - They had not reached line rate in any tests I was running
 - Routing showed that the stack consumed about 25% of the total
 - Thus I end up with 3 CPUs RX, 3 CPUs TX, and 2 CPUs stack



Results, Round 1

IXGBE Performance



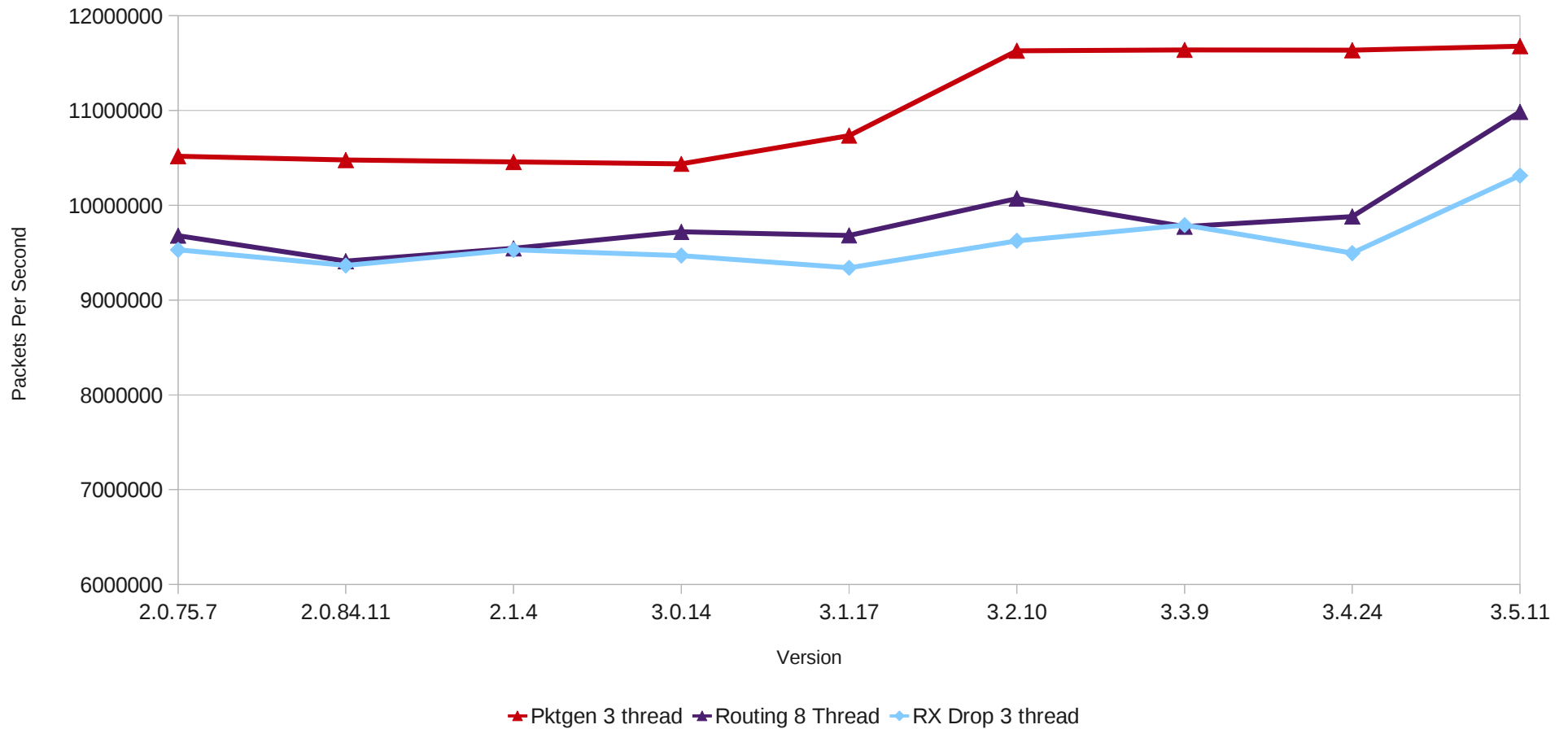
Something doesn't seem right..

- 2.1.4
 - Dropped support for RSS w/ UDP ports
 - This changed the work distribution
- 3.0.14
 - Removed trans_start from Tx path
 - This change was pushed upstream over a year prior
- 3.2.10
 - Removed last_rx from Rx path
 - Another change that made it upstream over a year prior



Results, Round 2

IXGBE Performance



Performance root cause

- 3.1.17
 - Set `TXDCTL.PTHRESH` to 32, allowing hardware to prefetch descriptors in groups of 8.
- 3.2.10
 - Combined all hotpath items in adapter struct into a single read-mostly cacheline
- 3.5.11
 - Enabled `SRRCTL.DROP_EN` when RX multiqueue is enabled and flow control is disabled



Cutting the memory overhead

- Combine all adapter fields accessed in hot-path into single cache-line to prevent cache pollution
- Configure hardware to batch descriptor reads & writes



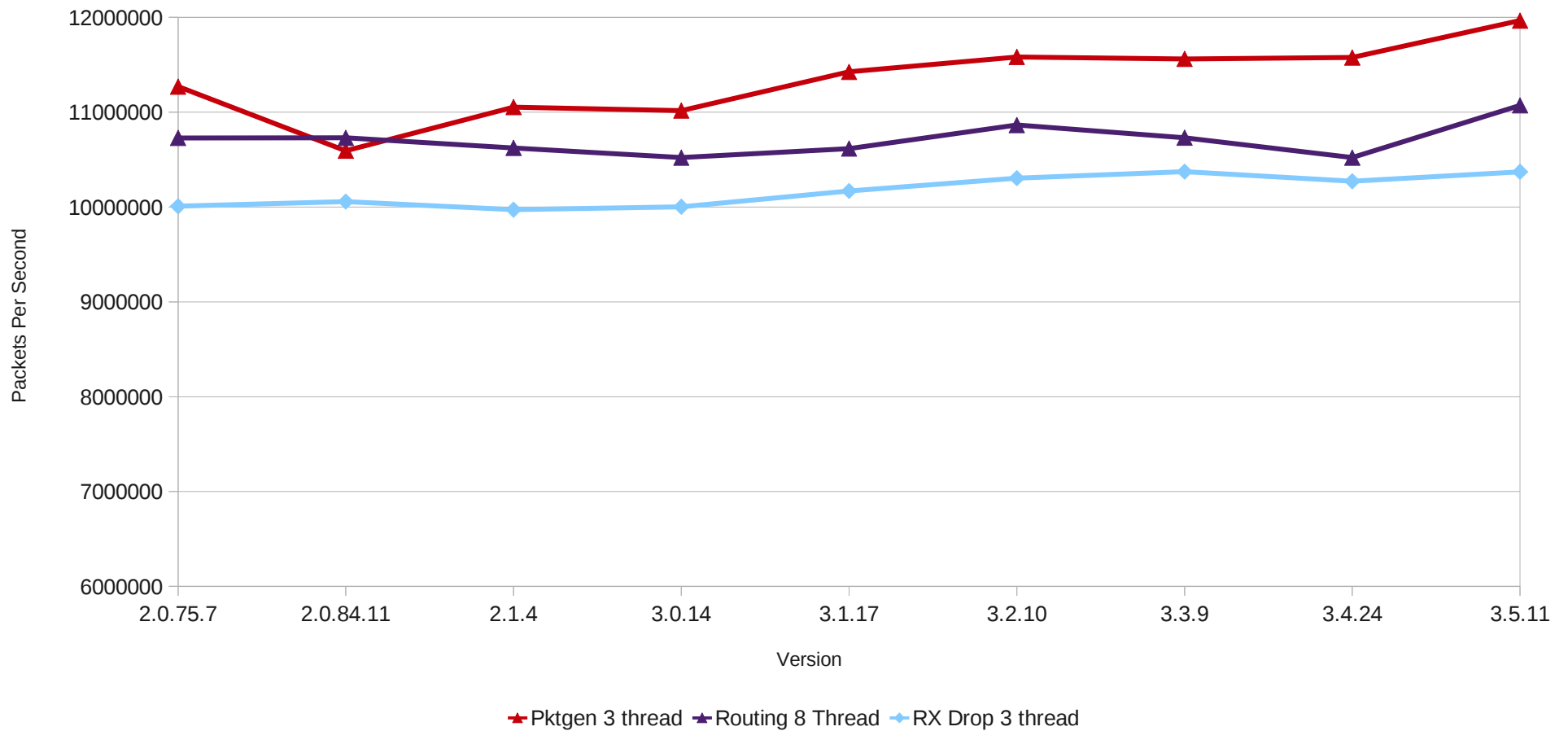
Knowing when to drop a packet

- Rx FIFO is yet another buffer that can introduce delays
 - Prone to head of line blocking in multiqueue configurations
 - Can only move as fast as the quickest ring
- `SRRCTL.DROP_EN` drops packets from Rx FIFO
 - Only drops packets when no buffers are available on ring to DMA to
 - Allows faster rings to keep processing while slower rings drop packets
 - Reduces overall dropped packet rate
 - Mutually exclusive with features like flow control and DCB



Results, Round 3

IXGBE Performance



Performance root cause

- 2.0.84.11
 - Performance regression due to alignment change of `ixgbe_clean_tx_irq` from 64 byte to 16 byte
- 3.1.17
 - Combined `ixgbe_tx_map` and `ixgbe_tx_queue` calls into a single function
 - Fused all NAPI cleanup into `ixgbe_poll`
 - General cleanup of TX and RX path
- 3.5.11
 - Store values in “first” `tx_buffer_info` struct sooner
 - Avoid unnecessary modification of TX descriptor in cleanup



Reduce & reuse to cut memory usage

- Store values in the Tx `buffer_info` structure instead of in the stack
- Make Tx/Rx cleanup paths leave descriptors rings untouched until new buffers are available
- Reduce memory reads in Tx path by separating read-mostly and write-mostly parts of the ring structure
- Allocate memory on local node to reduce memory access time
- Allocate `sk_buff` such that `skb->head` is fully used
sizes 512, 1.5K, 3K, 7K, & 15K are optimal
- Use only $\frac{1}{2}$ of a page to allow for page reuse via page user count



Reducing code complexity

ixgbe-2.0.75.7

ixgbe_msix_clean_tx
ixgbe_msix_clean_rx
ixgbe_msix_clean_many

ixgbe_poll
ixgbe_clean_txonly
ixgbe_clean_rxonly
ixgbe_clean_rxtx_many

ixgbe-3.5.11

ixgbe_msix_clean_rings

ixgbe_poll

- Avoid unnecessary duplication of effort in maintenance.
- By only having one `ixgbe_poll` call the compiler can optimize by in-lining all of the various functions used by the call.



Where do we still have room to improve?

- At this point ixgbe consumes only about 15% of the total CPU utilization
- Duplicate overhead in ixgbe_poll and __alloc_skb due to cache misses while zeroing & reading skb header
- 90% of spin lock overhead appears to be due to Qdisc lock taken in sch_direct_xmit

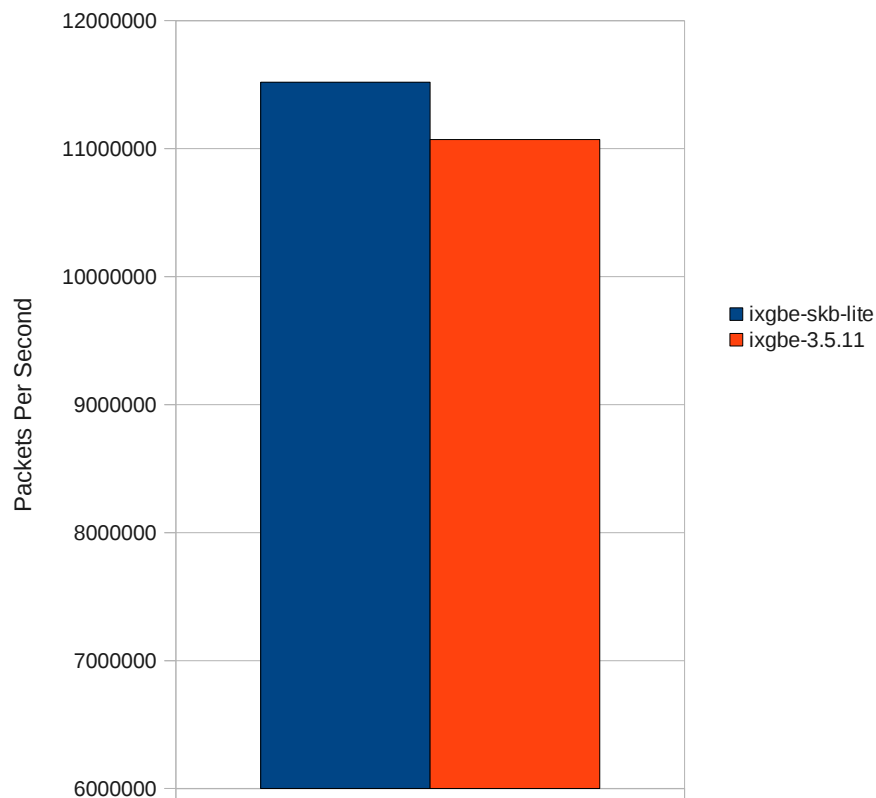
20.14%	[k]	_raw_spin_lock
7.76%	[k]	ixgbe_poll
6.58%	[k]	eth_type_trans
4.69%	[k]	__alloc_skb
4.68%	[k]	ip_rcv
4.36%	[k]	ixgbe_xmit_frame_ring
3.48%	[k]	ip_forward
3.37%	[k]	ip_route_input_common
3.30%	[k]	dev_queue_xmit
3.24%	[k]	kfree
3.03%	[k]	__netif_receive_skb
2.81%	[k]	kmem_cache_free
2.70%	[k]	kmem_cache_alloc_node
2.56%	[k]	kmem_cache_alloc_node_notrace
1.93%	[k]	ixgbe_alloc_rx_buffers
1.93%	[k]	__phys_addr
1.73%	[k]	memcpy
1.73%	[k]	skb_release_data
1.32%	[k]	ixgbe_select_queue
1.30%	[k]	dev_hard_start_xmit
1.26%	[k]	swiotlb_dma_mapping_error
1.14%	[k]	ip_finish_output
1.08%	[k]	__kmalloc_node
1.03%	[k]	local_bh_enable



What if we delay skb init?

Routing Performance

__alloc_skb split in two



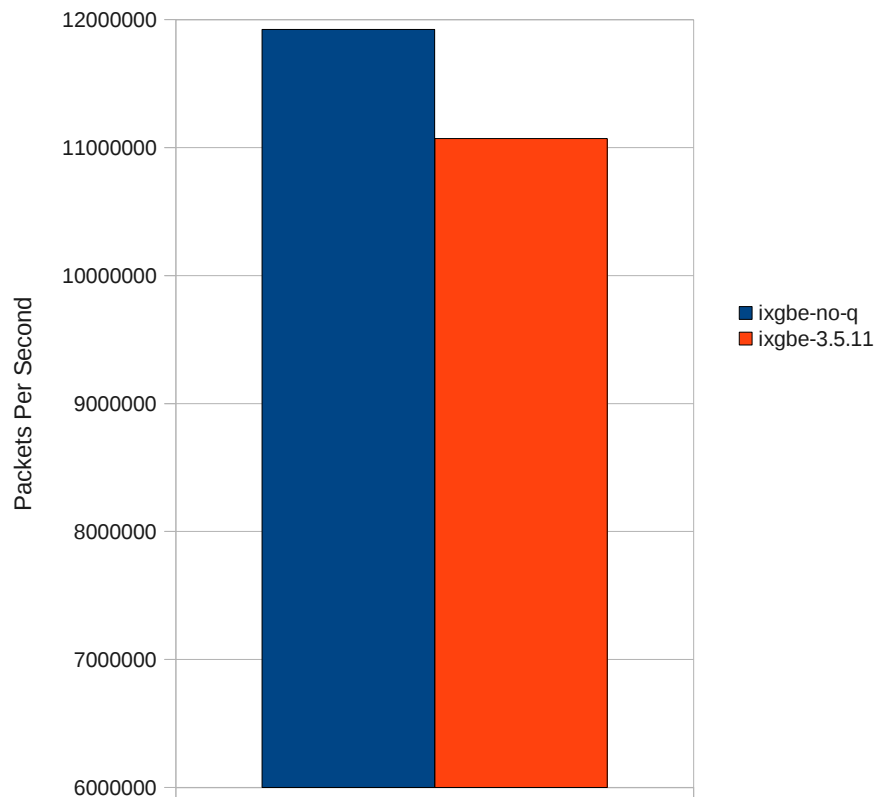
20.84%	[k] _raw_spin_lock
8.56%	[k] ixgbe_poll
5.38%	[k] ixgbe_xmit_frame_ring
4.40%	[k] ip_rcv
3.90%	[k] eth_type_trans
3.85%	[k] kfree
3.83%	[k] dev_queue_xmit
3.45%	[k] ip_route_input_common
2.98%	[k] kmem_cache_free
2.89%	[k] __netif_receive_skb
2.83%	[k] kmem_cache_alloc_node_notrace
2.68%	[k] kmem_cache_alloc_node
2.25%	[k] ip_forward
2.24%	[k] __phys_addr
1.79%	[k] memcpy
1.78%	[k] ixgbe_alloc_rx_buffers
1.57%	[k] is_swiotlb_buffer
1.52%	[k] dev_hard_start_xmit
1.36%	[k] init_skb_lite
1.36%	[k] skb_release_data
1.33%	[k] swiotlb_map_page
1.14%	[k] ip_finish_output
1.11%	[k] __alloc_skb_lite
1.10%	[k] local_bh_enable



What if we didn't have a Qdisc?

Routing Performance

With and Without Qdisc



22.26%	[k] ixgbe_poll
5.95%	[k] ip_rcv
5.12%	[k] __alloc_skb
4.97%	[k] ixgbe_xmit_frame_ring
4.54%	[k] __netif_receive_skb
3.78%	[k] ip_route_input_common
3.69%	[k] kfree
3.62%	[k] kmem_cache_alloc_node
3.48%	[k] ip_forward
3.26%	[k] kmem_cache_free
2.78%	[k] kmem_cache_alloc_node_notrace
2.69%	[k] eth_type_trans
2.49%	[k] dev_queue_xmit
2.23%	[k] __phys_addr
2.14%	[k] ixgbe_alloc_rx_buffers
1.91%	[k] memcpy
1.69%	[k] _raw_spin_lock
1.68%	[k] skb_release_data
1.51%	[k] dev_hard_start_xmit
1.35%	[k] swiotlb_map_page
1.21%	[k] ip_finish_output
1.14%	[k] __kmalloc_node
1.07%	[k] swiotlb_dma_mapping_error
1.01%	[k] is_swiotlb_buffer



