Memory vs. Networking

Provoking and fixing memory bottlenecks

Jesper Dangaard Brouer
Principal Engineer, Red Hat

Date: October 2016
Venue: NetConf, Tokyo, Japan

Memory vs. Networking

- Network provoke bottlenecks in memory allocators
  - Lots of work needed in MM-area
- Both in
  - kmem_cache (SLAB/SLUB) allocator
    - (bulk API almost done, more users please!)
  - Page allocator
    - Baseline performance too slow (see later graphs)
    - Drivers: page recycle caches have limits
      - Does not address all areas of problem space

<u>MM:</u> Status on kmem_cache bulk

- Discovered IP-forwarding: hitting slowpath
  - in kmem_cache/SLUB allocator
- Solution: Bulk APIs for kmem_cache (SLAB+SLUB)
  - Status: upstream since kernel 4.6
  - Netstack use bulk *free* of **SKB**s in NAPI-context
    - Use bulking opportunity at DMA-TX completion
    - 4-5% performance improvement for IP forwarding
  - Generic kfree_bulk API
- Rejected: Netstack bulk *alloc* of SKBs
  - As number of RX packets were unknown

MM: kmem_cache bulk, more use-cases

- Network stack – more use-cases
  - Need explicit bulk free use from TCP stack
    - NAPI bulk free, not active for TCP (keep ref too long)
  - Use kfree_bulk() for skb→head
    - (when allocated with kmalloc)
  - Use bulk free API for qdisc delayed free
- RCU use-case
  - Use kfree_bulk() API for delayed RCU free
- Other kernel subsystems?

SKB overhead sources

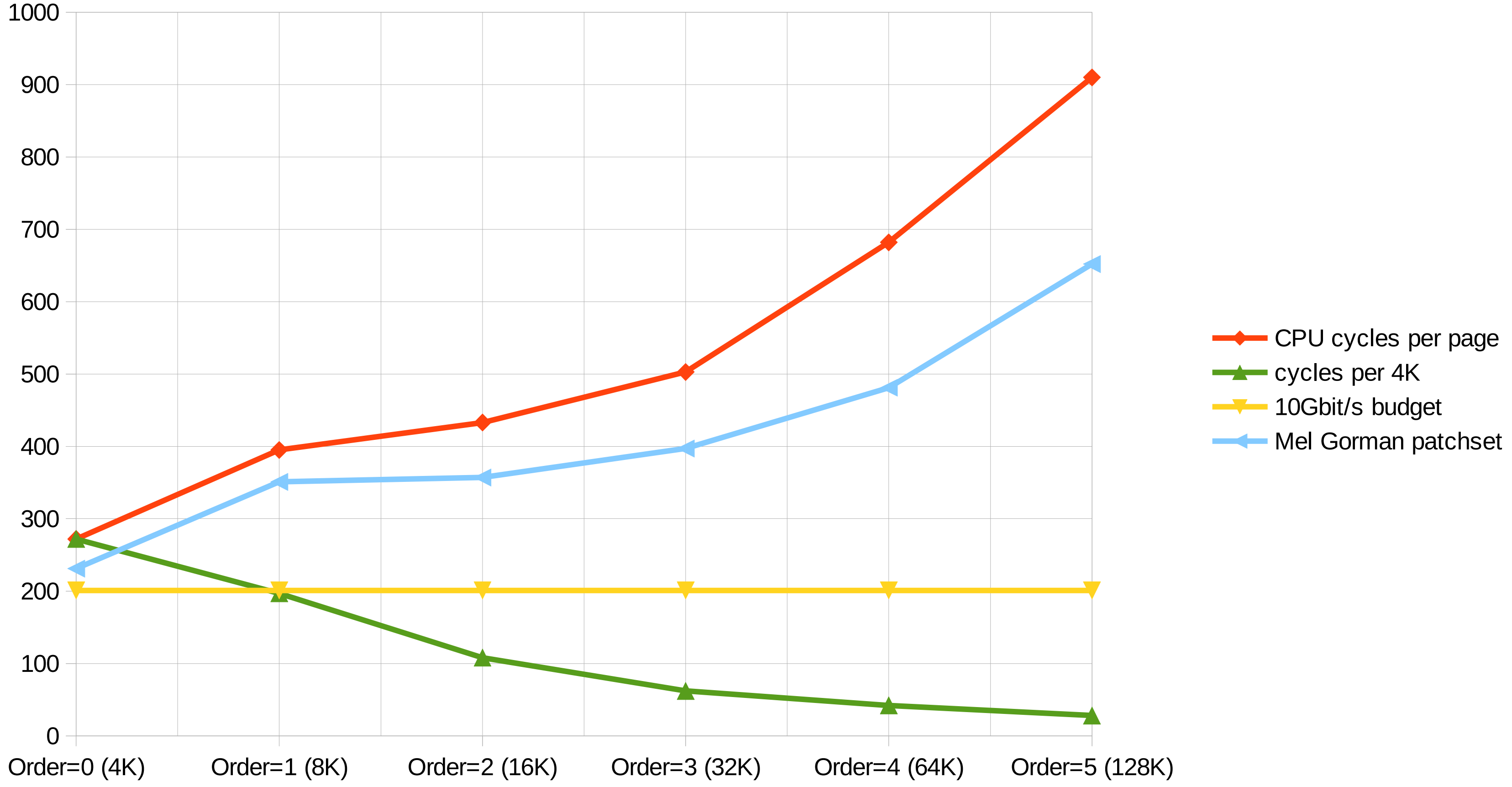
- Sources of overhead for SKBs (`struct sk_buff`)
  - Memory alloc+free
    - Addressed by kmem_cache bulk API
  - Clearing SKB
    - Need to clear 4 cache-lines!
  - Read-only RX pages
    - Cause more expensive construction the SKB

SKB allocations: with read-only pages

- Most drivers have read-only RX pages
  - Cause more expensive SKB setup
    1) Alloc separate writable mem area
    2) memcpy over RX packet headers
    3) Store skb_shared_info in writable-area
    4) Setup pointers and offsets, into RX page-"frag"
- Reason: Performance trade off
    A) Page allocator is too slow
    B) DMA-API expensive on some platforms (with IOMMU)
    - Hack: alloc and DMA map larger pages, and “chop-up” page
    - Side-effect: read-only RX page-frames
      - Due to unpredictable DMA unmap time

Benchmark: Page allocator (optimal case, 1 CPU, no congestion)

- Single page (order-0) too slow for 10Gbit/s budget
- Cycles cost increase with page order size
  - But partitioning page into 4K fragments amortize cost

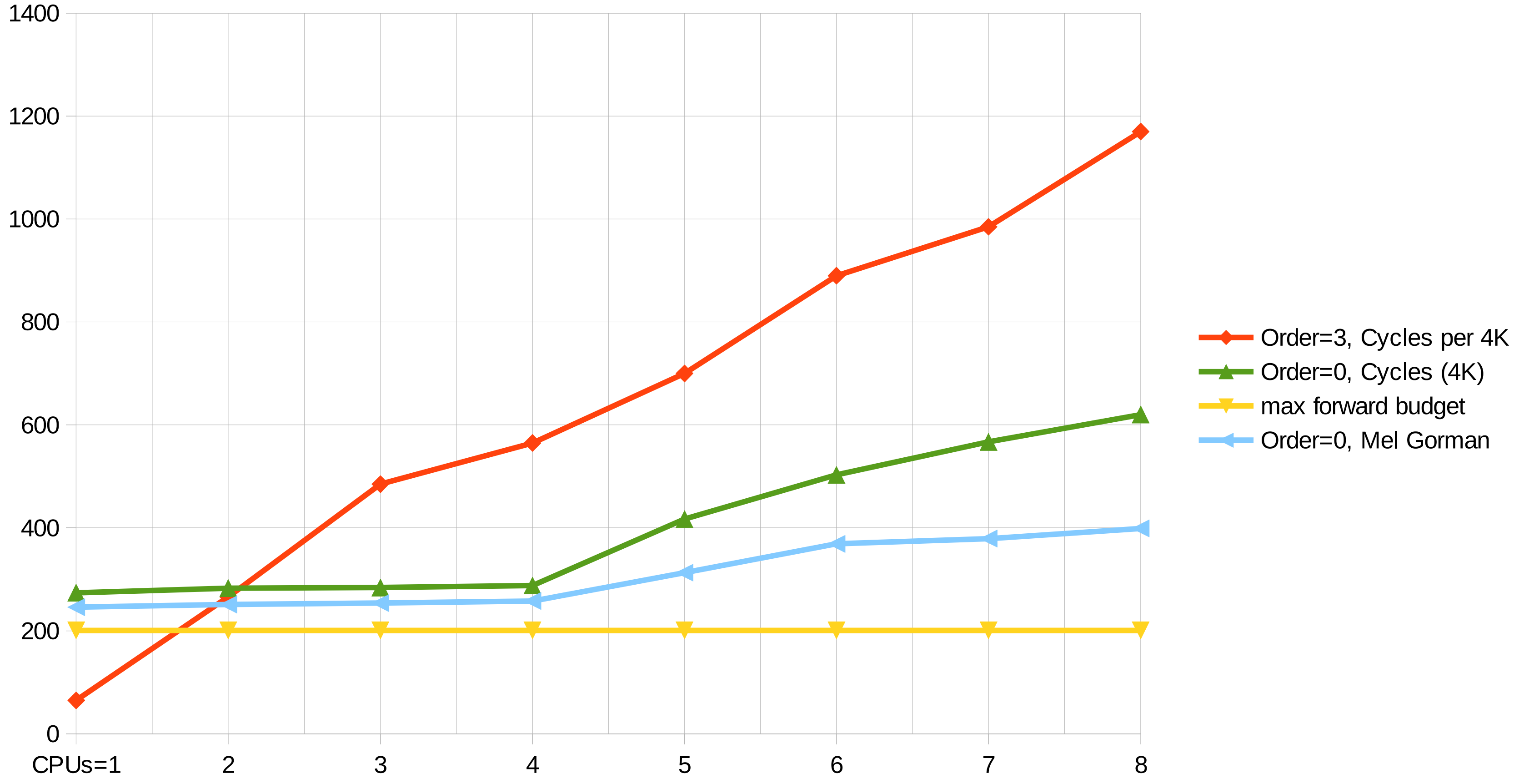


Issues with: Higher order pages

- Performance workaround:
  - Alloc larger order page, handout fragments
    - Amortize alloc cost over several packets
- Troublesome
  - 1. fast sometimes and other times require reclaim/compaction which can stall for prolonged periods of time.
  - 2. clever attacker can pin-down memory
    - Especially relevant for end-host TCP/IP use-case
  - 3. does not scale as well, concurrent workloads

Concurrent CPUs scaling micro-benchmark

- Danger of higher order pages, with parallel workloads
  - Order=0 pages scale well
  - Order=3 pages scale badly, even counting per 4K
    - Already lose advantage with 2 concurrent CPUs



<u>RX-path:</u> Make RX pages writable

- Need to make RX pages writable
- Why is page (considered) read-only?
  - Due to DMA_unmap time
    - Several page fragments (packets) in-flight
    - Last fragment in RX ring queue, call dma_unmap()
    - DMA engine unmap semantics allow overwriting memory
      - (Not a problem on Intel)
- Simple solution: Use one-packet per page
  - And call dma_unmap before using page
- My solution is the page_pool

<u>Page pool:</u> Generic solution, many advantages

- 5 features of a recycling page pool (per device):
  1) Faster than page-allocator speed
     - As a specialized allocator require less checks
  2) DMA IOMMU mapping cost removed
     - Keeping page mapped (credit to Alexei)
  3) Make page writable
     - By predictable DMA unmap point
  4) OOM protection at device level
     - Feedback-loop know #outstanding pages
  5) Zero-copy RX, solving memory early demux
     - Depend on HW filters into RX queues

Page pool: Design

- Idea presented at MM-summit April 2016
- Basic concept for the page_pool
  - Pages are recycled back into originating pool
    - Creates a feedback loop, helps limit pages in pool
  - Drivers still need to handle dma_sync part
  - Page-pool handle dma_map/unmap
    - essentially: constructor and destructor calls
- Page free/return to page-pool, Either:
  1) SKB free knows and call page pool free, **or**
  2) put_page() handle via page flag

<u>Page-pool:</u> opportunity – feedback loop

- Today: Unbounded RX page allocations by drivers
  - Can cause OOM (Out-of-Memory) situations
  - Handled via skb->truesize and queue limits
- Page pool provides a **feedback loop**
  - (Given pages are recycles back to originating pool)
  - Allow bounding pages/memory allowed per RXq
    - Simple solution: configure fixed memory limit
    - Advanced solution, track steady-state
      - Can function as a “Circuit Breaker” (See RFC draft link)

SKB clearing cost is high

- Options for addressing clearing cost:
  - Smaller/diet SKB (currently 4 cache-lines)
    - Too hard!
  - Faster clearing
    - Hand optimized clearing: only save 10 cycles
    - Clear larger contiguous mem (during bulk alloc API)
  - Delay clearing
    - Don't clear on alloc (inside driver)
      - Issue: knowing what fields driver updated
    - Clear sections later, inside netstack RX
    - Allow prefetchw to have effect

<u>Topic: RX-MM-allocator</u> – Alternative

- <u>Prerequisite:</u> When page is writable
- Idea: No SKB alloc calls during RX!
  - Don't alloc SKB,
    - Create it inside head or tail-room in data-page
  - skb_shared_info, placed end-of data-page
  - Issues / pitfalls:
    1) Clear SKB section likely expensive
    2) SKB truesize increase(?)
    3) Need full page per packet (ixgbe does page recycle trick)

The end

- kfree_bulk(16, slides);