MM-summit 2016

# Generic page-pool recycle facility?

Jesper Dangaard Brouer
Principal Engineer, Red Hat

MM-summit 2016: April 18th-19th

# Intro slide: Motivation for page recycling

- Bottlenecks: in both page allocator and DMA APIs
  - Many driver specific workarounds
    - and unfortunate side-effect of workarounds
- Motivation(1): primarily performance motivated
  - Building “packet-page”/XDP level forward/drop facility
- Motivation(2): drivers are reinventing
  - Cleanup open-coded driver approaches?!
- Motivation(3): other use-cases
  - Like supporting zero-copy RX

# Optimization principle behind page-pool idea

- Untapped optimization potential
  - Recycling pages,
    - instead of always returning to page allocator
  - Opens up for a number of optimizations, in area
    - shifting computation and setup time,
    - to when enter/leaving pool

# DMA bottleneck: mostly on PowerPC

- On arch's like PowerPC: DMA API is the bottleneck
- Driver work-around: amortize dma call cost
  - alloc large order (compound) pages.
    - **dma_map** compound page, handout **page-frag**ments for RX ring, and later **dma_unmap** when last RX page-fragments is seen.
- Bad side-effect: DMA page considered 'read-only'
    - Because dma_unmap call can be destructive
      - NOP instruction on x86
  - Read-only side-effect: Cause netstack overhead:
    - alloc new writable memory, copy-over IP-headers, and adjust offset pointer into RX-page

# Idea to solve DMA mapping cost (credit Alexei)

- Keep these pages DMA mapped to *device*
  - page-pool is recycling pages
  - ***back to*** *the originating* ***device***
- Avoid the need to call dma_unmap
  - Only call dma_map() when setting up pages
  - And DMA unmap when leaving pool
- This should solve both issues
  - Removed cost of DMA map/unmap
  - Can consider DMA pages writable
    - (dma_sync determine when)

# DMA trick: “Spelling it out”

- For DMA “keep-mapped-trick” to work
  - Pages must be return to originating device
    - To make “static” DMA map valid
- Without storing info in struct-page
  - Troublesome to track originating device
    - Needed at TX DMA completion time of another device
  - (also track DMA unmap addr for PowerPC)
- Any meta-data to track originating device
  - Cannot be free'ed until after TX DMA
- Could use page→private

# Page allocator too slow

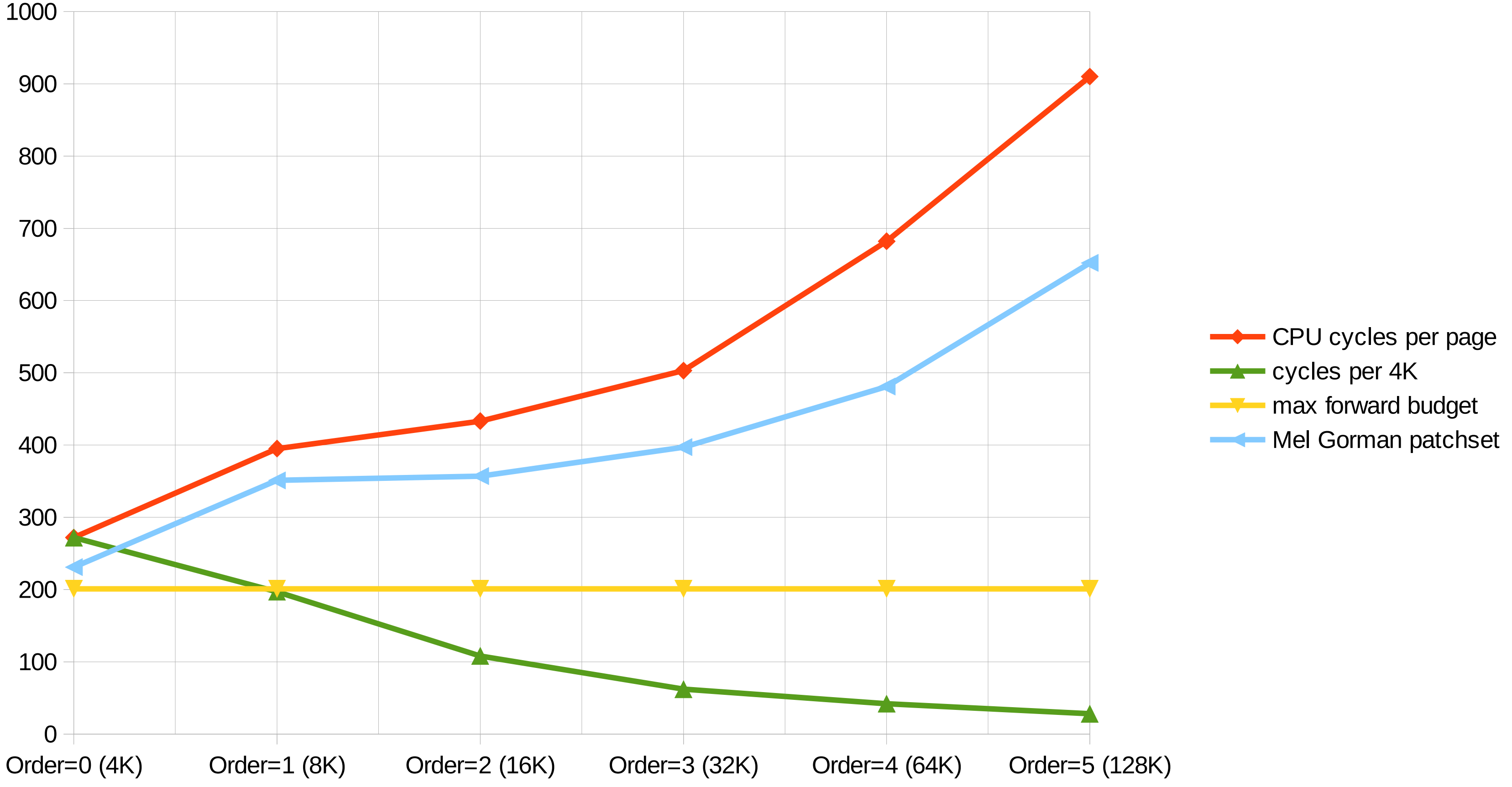
- On x86, DMA is NOT the bottleneck
  - Besides the side-effect of read-only pages
- XDP (eXpress Data Path) performance target
  - 14.8 Mpps, approx 201 cycles at 3GHz
- Single page order-0: cost 277 cycles
  - alloc_pages() + __free_pages()
    - (Mel's patchset reduced this to: 231 cycles)

# Work around for slow page allocator

- Drivers use: same trick as DMA workaround
  - Alloc larger order page: And handout fragments
- E.g. Page order-3 (32K): cost 503 cycles (Mel 397 cycles)
  - Handout 4K blocks, cost per block: 62 cycles
- Problematic due do memory pin down “attacks”
  - Google disable this driver feature
- See this as a bulking trick
  - Instead implement a page bulk API?

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

- Cycles cost increase with page order size
  - But partitioning page into 4K fragments amortize cost

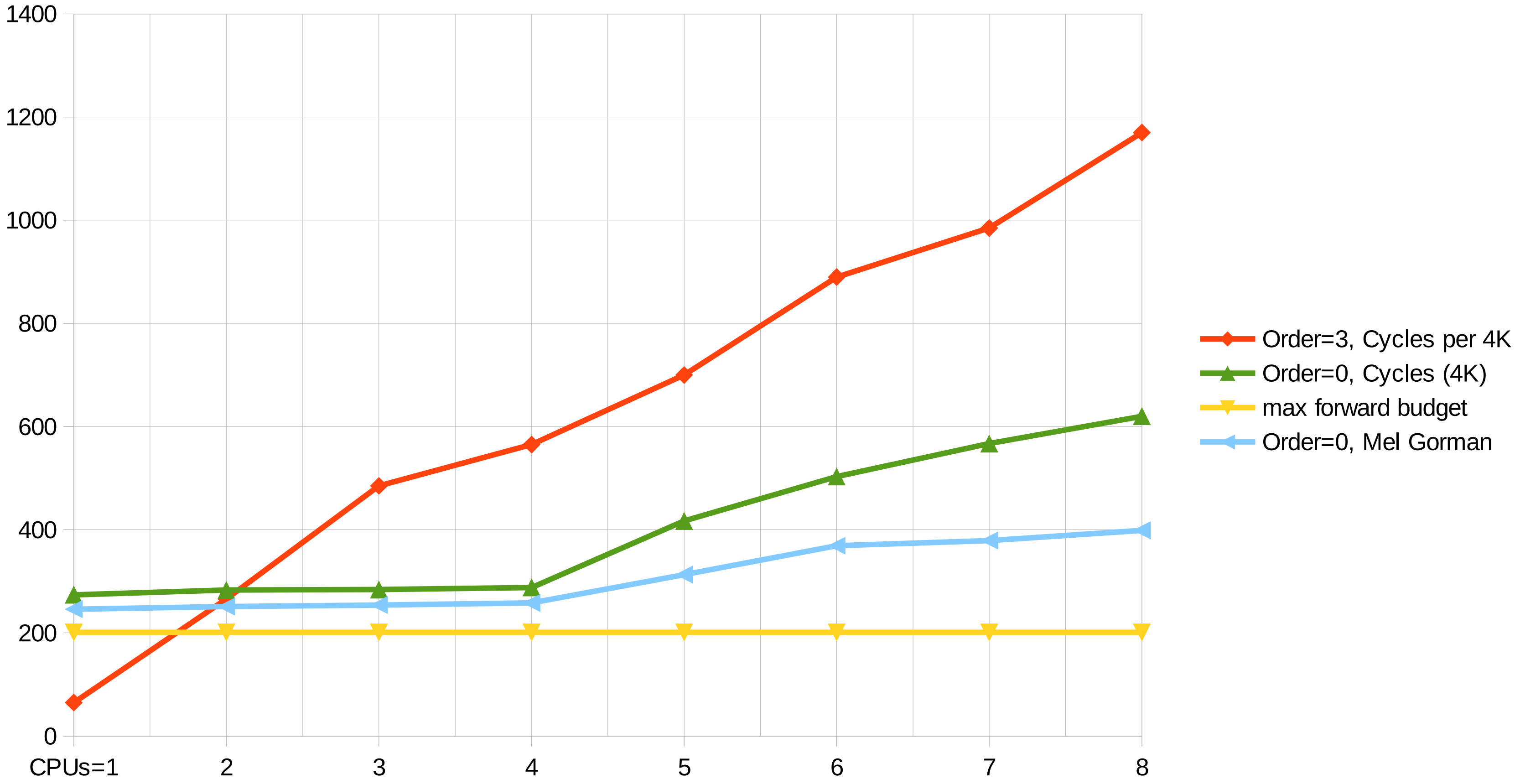


# Issues with: Higher order pages

- Hidden bulking trick
    - Alloc larger order page, handout fragments
- 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

- Order=0 pages scale well
- Order=3 pages scale badly, even counting per 4K
  - Already lose advantage with 2 concurrent CPUs



# Page-pool cooperating

- Avoid keeping too many pages
- Steady state, RX=TX rate, no queue
  - Only requires RX ring size + TX DMA outstanding
  - Thus, restrict pool size can be small
- Overload/Burst state, RX > TX rate, cause queue
  - “Good queue” behavior absorb bursts
  - “Bad queue” (long standing queue) potential for OOM
    - Today: handled at different levels, socket queue limit
    - Potential for detecting “bad queue” at this level
- Allow page allocator to reclaim pool pages

# Big question: How integrated with MM-layer

- Big “all-in” approach:
  - Become allocator like slub: use struct page
  - Minimum: page pointer back to page_pool
    - And DMA unmap address
- Build as shell around page allocator
  - How to keep track of “outstanding” pages?
    - + track DMA unmap addr per page
  - API users keep track of which pool to return to
    - At TX completion time, return info needed
      - Thus, meta-data is kept around too long (cache-cold)
    - Might be a trick to avoid this, by sync on page refcnt

# Novel recycle trick by Intel drivers

- Issue getting page recycled back into pool
  - Without meta-data keeping track of return-pool
- Use page ref count
  - To see if TX is done, when RX look at page
  - Split pages in two halfs
    - Keep pages in RX ring (tracking structure)
    - On RX, if page refcnt is low (<=2),
      - then reuse other half to refill RX ring (else normal alloc)
    - In-effect recycle the page
      - When one-time round ring is less than TX complet time
  - Still, adds 2x atomic ops per packet

# Other use-cases: RX zero-copy

- Currently: NIC RX zero-copy not allowed
  - Could leak kernel memory information in page
- Know: Pages are recycled back into pool
  - Clear memory on new page entering pool
  - RX zero-copy safe, but could “leak” packet-data
- Early demux: HW filters can direct to specific RX-q
  - Create page-pool per RX-queue
  - Idea: alloc pages from virtual addr space (premapped)
- Need fairly closer integration with MM-layer
  - (not compatible with Intel driver trick)

# Other use-cases: Using huge pages for RX

- Make page-pool API hide page-boundaries
  - Driver unaware of page order used
- Idea: huge page RX zero-copy
  - Page-pool handout page-frags for RX ring
  - Huge-page gets memory mapped into userspace
    - Done to reduce TLB misses for userspace
  - Zero-copy to userspace
    - Netmap or DPDK could run on top
- Use NIC HW filter,
  - create RX queue with this pool strategy
- Hardlimit on number huge pages

# Concluding discussion!?

- Already active discussions on mailing list…
- Must fix DMA problem causing read-only pages
  - Maybe just have “ugly” solution for x86?
- Leaning towards, something on top of page allocator
  - Only focus on performance use-case
  - Down prioritize RX zero-copy use-case?
  - Use field in struct page, for pool return path
- Want a page bulk alloc API… please!
  - For faster refill of page-pool
  - Large order page trick is problematic (as described)