Memory vs. Networking

Provoking and fixing memory bottlenecks

Focused on the page allocator

Jesper Dangaard Brouer
Principal Engineer, Red Hat

LSF/MM-summit
March, 2017

Overview

- Mind-bugling speeds
- New technology: XDP (eXpress Data Path)
  - Limited by page allocator speed
- Page recycling
  - Every driver reinvent page recycling
  - Generalizing page recycling
- Micro benchmarking page allocator
  - Huge improvements by Mel Gorman
  - Strange overhead by NUMA

Mind-bugling speeds

- 10Gbit/s wirespeed smallest packet size
  - 14.88 Mpps (Million packets per second)
  - 67.2ns between packets, at 3GHz -> 201 cycles
  - 100G with 1514 bytes pkt size → 8.1Mpps (123ns)
- Trick: Bulking/batching: amortize per packet cost
  - Easy to think: bulk 10, 10x cycle budget 2010 cycles.
    - Not as easy: bulking APIs does not give linear scaling
    - E.g. SLUB bulking APIs gave 60% speedup.
- Kernel bypass solutions show it is possible!
  - Via bulking and own memory allocators

XDP - eXpress Data Path

- NetDev-guys XDP technology (approx kernel v4.9)
  - Shows these speeds are possible, as long as:
    - 1. we avoid talking to MM-layer
    - 2. page kept DMA mapped
    - 3. stay on same CPU
- Single CPU performance (mlx5 50Gbit/s)
  - XDP_DROP: 17Mpps
  - XDP_TX: 10Mpps (TX out same interface)
- Use-cases:
  - DDoS and Load-Balancer (Facebook)

XDP “real” forwarding missing

- XDP packet forward between devices
  - Not implemented yet,
    - due to performance concerns and missing RX bulking
  - Cannot avoid interacting with MM-layer
    - Local device driver specific recycle trick not sufficient
- Imagined forwarding to another device
  - No-SKB, “raw” page is transferred, offset+length
  - Sits on remote device TX queue
    - Until DMA TX completion: Now page need free’ed or “returned”

Driver page recycling

- All high-speed NIC drivers do page recycling
  - Two reasons:
    - 1. page allocator is too slow
    - 2. Avoiding DMA mapping cost
- Different variations per driver
  - Want to generalize this
    - Every driver developer is reinventing a page recycle mechanism

Need “handle” to page after leaving the driver

- Some drivers do opportunistic recycling today
  - Bump page refcnt, and keep pages in a queue
    - (Intel drivers use RX ring itself for this queue)
  - On alloc, check if queue-head page have refcnt ==1
    - If so, reuse this
    - Else, remove from queue and call put_page()
      - Thus, last caller of put_page() will free it for real
      - Issue: call DMA-unmap on pkts in-flight
- This looks good in benchmarks, but will it work for:
  - Real use-cases: many sockets that need some queue
  - TCP sockets: keep packets for retransmit until ACKed

Generalize problem statement

- Drivers receive DMA mapped pages
  - Want to keep page DMA mapped (for perf reasons)
  - Thinks page allocator too slow
- What can MM layer do address this use-case?
  - Faster PCP (Per CPU Page) cache
  - But can this ever compete with driver local recycling
  - XDP_DROP return page into array (no-locks)
    - (protected by NAPI/softirq running)
- Could MM provide API for
  - per device page allocator (limited size) cache
    - that keeps pages DMA mapped for this device

Even more generic

- What I'm basically asking for: destructor callback
  - Upon page reach refcnt == 0
    - (+separate call to allow refcnt==1 when safe)
  - Call a device specific destructor callback
  - This call is allowed to steal the page
  - callback gets page + data (in this case DMA address)
- Road blocks
  - Need page-flag
  - Room to store
    - Data (DMA-addr) + Callback (can be table lookup id)
      - (looking at page->compound_dtor infrastructure)

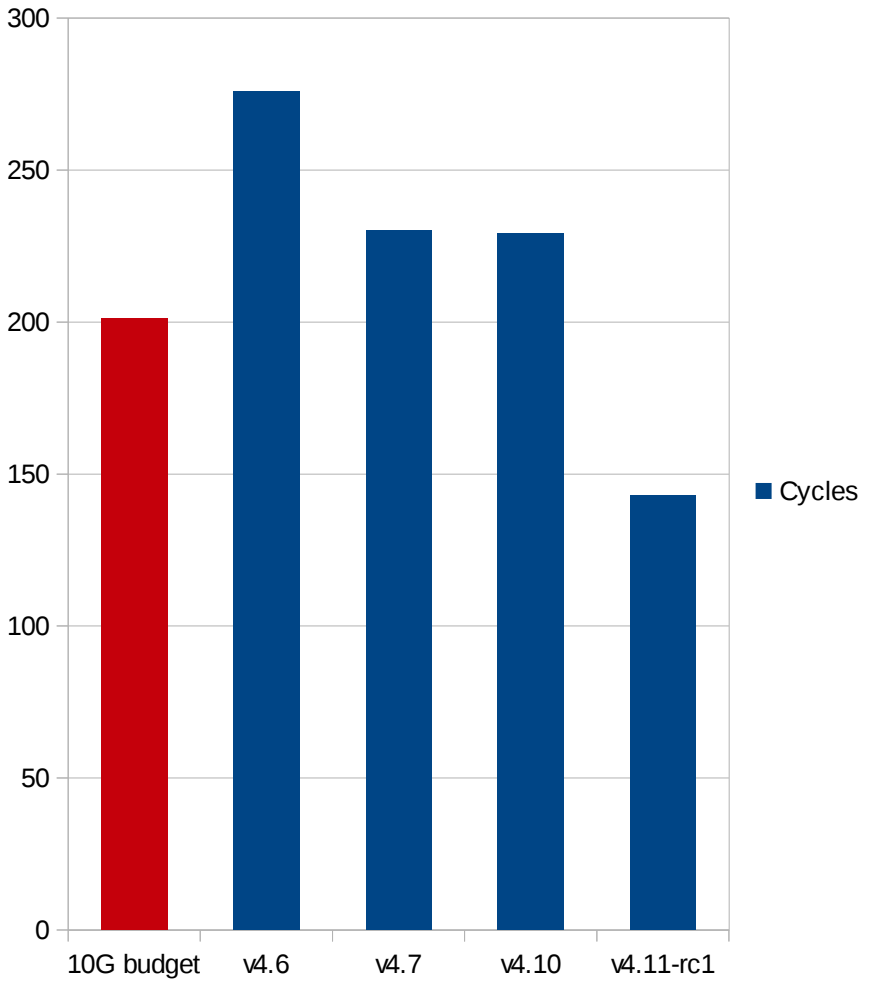
Microbenchmarking time

- Quote Sir Kelvin:
  - "If you cannot measure it, you cannot improve it"
- What is the performance of the page allocator?
  - How far are we from Jesper’s target?

- **Do be aware:** This is "zoom-in" microbenchmarking, designed to isolate and measure a specific code path, magnifying bottlenecks.
  - **Don’t get scared:** Provoked lock congestion’s should not occur for real workloads

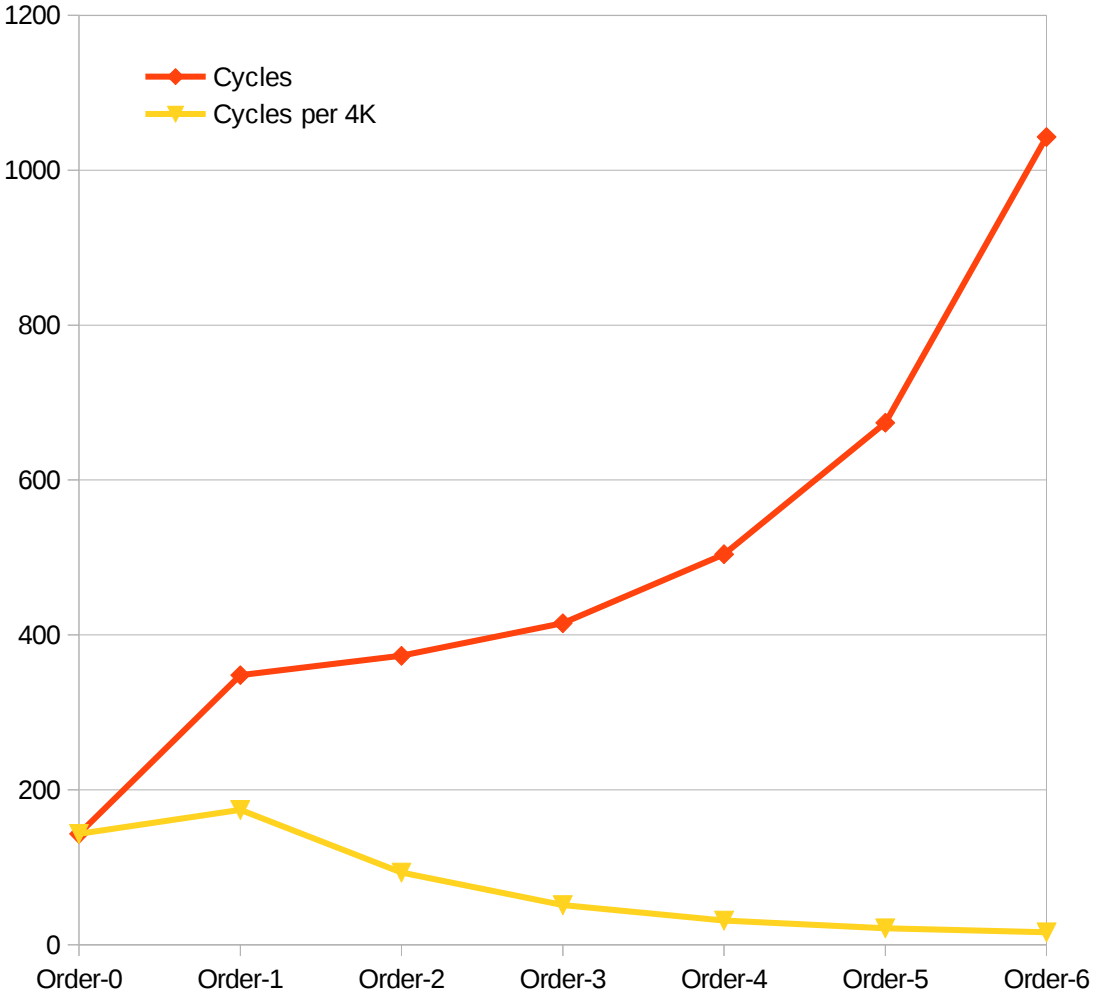
History: Benchmark page order-0 fast-path

- Micro-benchmark order-0 fast-path
  - Test PCP (Per CPU Pages) lists
    - Simply recycle same page
    - Only show optimal perf achievable
- Graph show perf improvement history
  - All credit goes to **Mel Gorman**
  - Approx 48% improvement!!! :-)
    - 276 → 143 cycles



Cost when page order increase (Kernel 4.11-rc1)

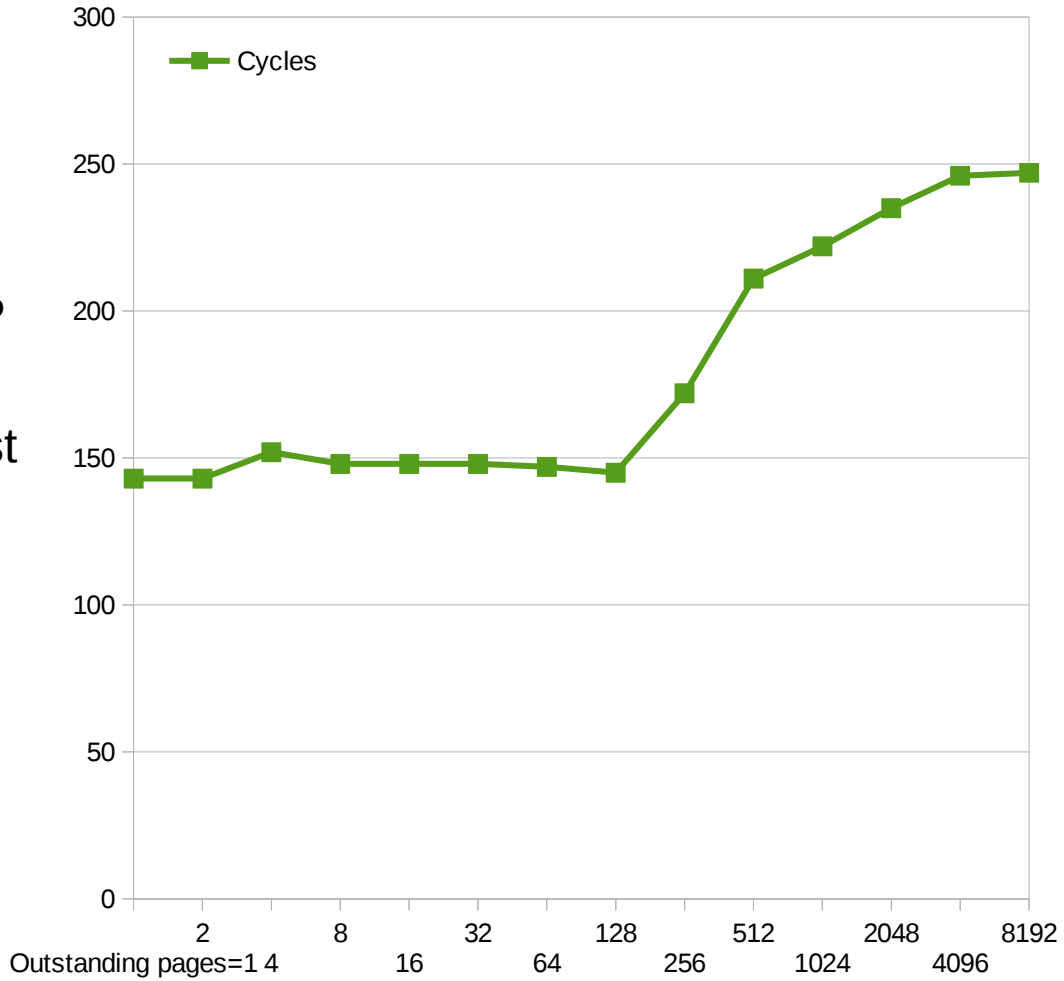
- Redline no surprises here
  - Expected:
    - Cost goes up with order
  - Expected:
    - Good curve until order-3
- Yellow line
  - Amortize cost per 4K
  - Trick used by some drivers
  - Want to avoid this trick:
    - Attacker pin down memory
    - Bad for concurrent workload
    - Reclaim/compaction stalls



Pressure PCP lists with many out-standing pages

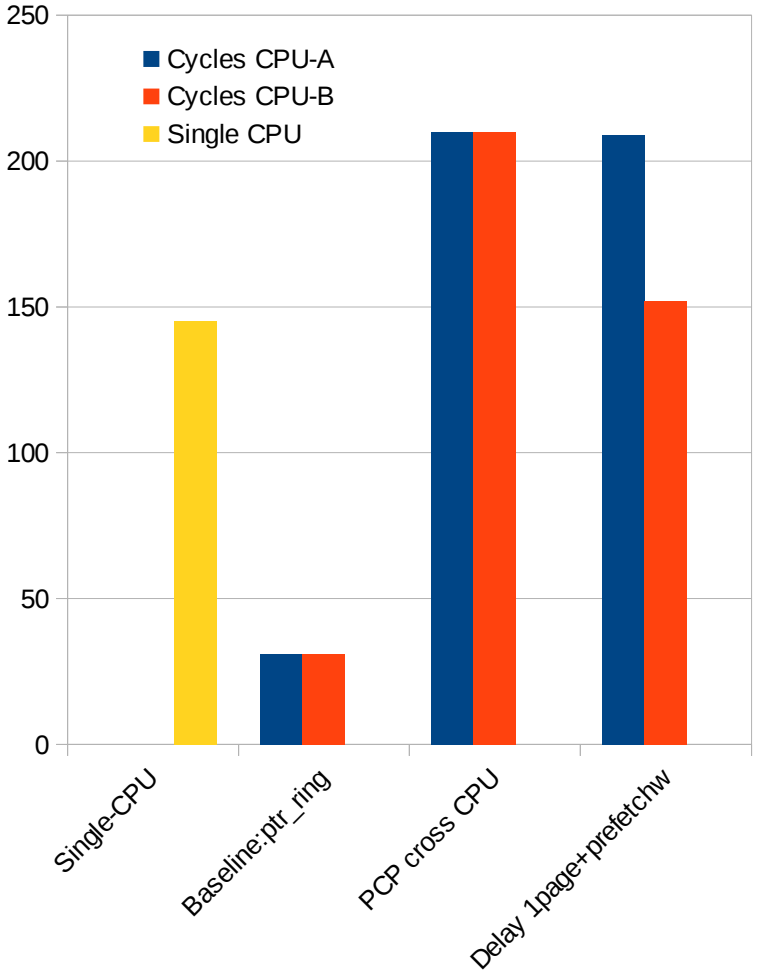
- Test alloc N order-0 pages before freeing them
  - Expected results:
    - Clearly shows size of PCP cache is 128
    - Fairly good at amotize cost

      (Does 32 bulking internally)



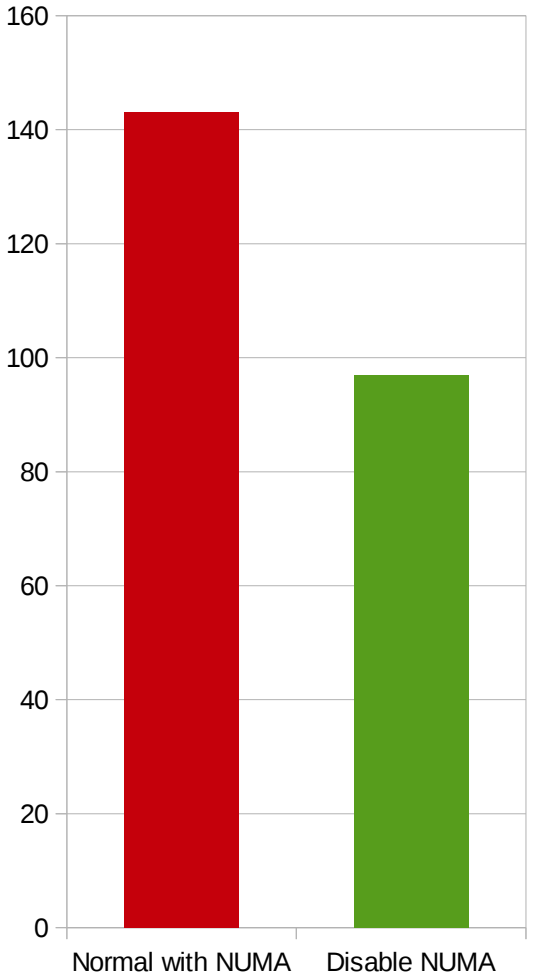
Moving pages cross CPU

- Networking often RX on CPU-A but process and free CPU-B
- Bench isolate page moving cross CPU
- Baseline: ptr_ring (31 cycles per CPU)
  - Best possible perf with cross CPU queue
- PCP cross CPU(order-0) alloc_pages+put_page
  - Cost on both CPUs increased to 210 cycles
    - Cross CPU issue? Or
    - outstanding pages issue? (210-31)*2=358 too high
    - Hot lock: zone→lock (even-though PCP have 32 bulking)
  - Single CPU cost 145 cycles (no outstanding)
- Delay 1 page + prefetchw before put_page()
  - CPU-B cost 152 reduced with 58 cycles
  - Helps CPU cache coherency protocol



Disable zone_statistics (via No-NUMA)

- Micro-benchmark order-0 fast-path
  - Test PCP (Per CPU Pages) lists
    - Simply recycle same page
    - Only show optimal perf achievable
- Disable CONFIG_NUMA and zone_statistics
  - (Kernel 4.11-rc1)
  - 143 cycles normal with NUMA
  - 97 cycles disabled NUMA
  - Extra 46 cycles cost (32%)
    - Looks like most originate from call
    - zone_statistics()



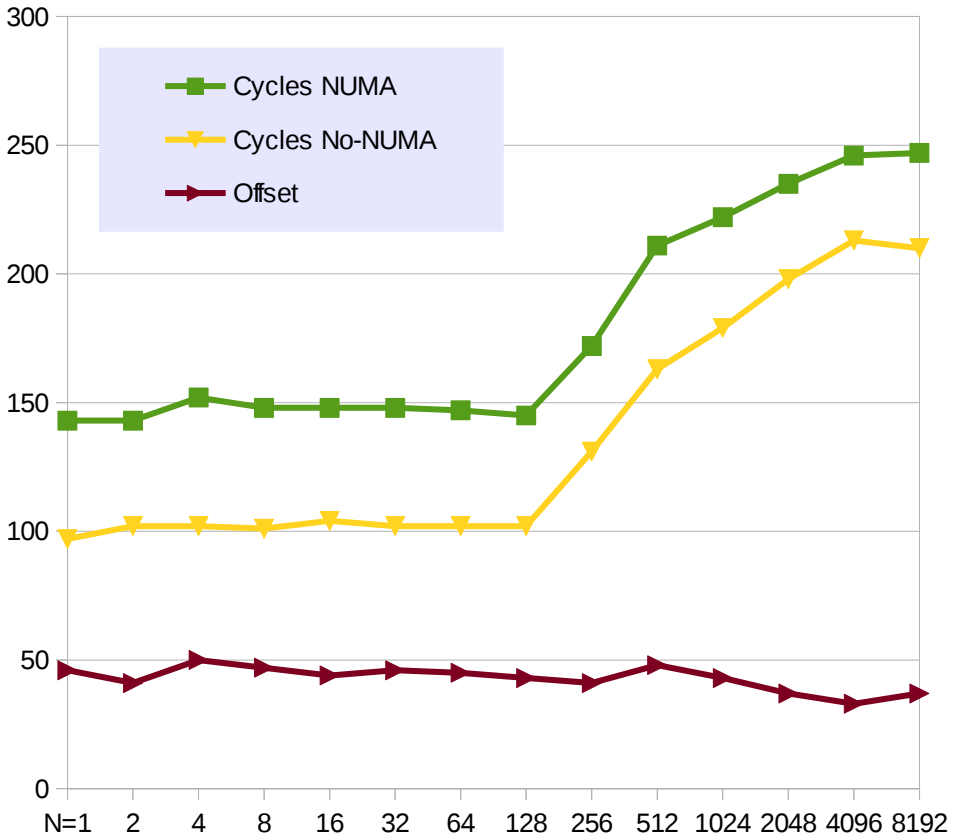
End slide

- Use-case: XDP redirect out another device
  - Need super fast return/free of pages
  - Can this be integrated into page allocator
    - Or keep doing driver opportunistic recycle hacks?

EXTRA SLIDES

Compare vs. No-NUMA out-standing pages test

- Test alloc N order-0 pages before freeing them
  - Expected results:
  - With or without NUMA
    - Still follow curve
    - Offset is almost constant



Page allocator bulk API benchmarks

- Took over some bulk patches from Mel
- Rebasing patches to latest kernel
  - Ran into issue… no results to present :-(
    - Fails in zone_watermark_fast check
  - Did write page_bench04_bulk