Page Pool API

The page_pool allocator is optimized for the XDP mode that uses one frame per-page, but it can fallback on the regular page allocator APIs.

Basic use involves replacing alloc_pages() calls with the page_pool_alloc_pages() call. Drivers should use page_pool_dev_alloc_pages() replacing dev_alloc_pages().

API keeps track of inflight pages, in order to let API user know when it is safe to free a page_pool object. Thus, API users must run page_pool_release_page() when a page is leaving the page_pool or call page_pool_put_page() where appropriate in order to maintain correct accounting.

API user must call page_pool_put_page() once on a page, as it will either recycle the page, or in case of refcnt > 1, it will release the DMA mapping and inflight state accounting.

Architecture overview

System Message: WARNING/2 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\[linux-master][Documentation][networking]page_pool.rst`, line 27)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

    +------------------+
    |       Driver     |
    +------------------+
            ^
            |
            |
            |
            v
    +--------------------------------------------+
    |                request memory              |
    +--------------------------------------------+
        ^                                  ^
        |                                  |
        | Pool empty                       | Pool has entries
        |                                  |
        v                                  v
    +-----------------------+     +------------------------+
    | alloc (and map) pages |     |  get page from cache   |
    +-----------------------+     +------------------------+
                                    ^                    ^
                                    |                    |
                                    | cache available    | No entries, refill
                                    |                    | from ptr-ring
                                    |                    |
                                    v                    v
                          +-----------------+     +------------------+
                          |   Fast cache    |     |  ptr-ring cache  |
                          +-----------------+     +------------------+
```

API interface

The number of pools created **must** match the number of hardware queues unless hardware restrictions make that impossible. This would otherwise beat the purpose of page pool, which is allocate pages fast from cache without locking. This lockless guarantee naturally comes from running under a NAPI softirq. The protection doesn't strictly have to be NAPI, any guarantee that allocating a page will cause no race conditions is enough.

- page_pool_create(): Create a pool.
  - flags: PP_FLAG_DMA_MAP, PP_FLAG_DMA_SYNC_DEV
  - order: 2^order pages on allocation
  - pool_size: size of the ptr_ring
  - nid: preferred NUMA node for allocation
  - dev: struct device. Used on DMA operations
  - dma_dir: DMA direction
  - max_len: max DMA sync memory size
  - offset: DMA address offset

- page_pool_put_page(): The outcome of this depends on the page refcnt. If the driver bumps the refcnt > 1 this will unmap the page. If the page refcnt is 1 the allocator owns the page and will try to recycle it in one of the pool caches. If PP_FLAG_DMA_SYNC_DEV is set, the page will be synced for_device using dma_sync_single_range_for_device().
- page_pool_put_full_page(): Similar to page_pool_put_page(), but will DMA sync for the entire memory area configured in area pool->max_len.
- page_pool_recycle_direct(): Similar to page_pool_put_full_page() but caller must guarantee safe context (e.g NAPI), since it will recycle the page directly into the pool fast cache.
- page_pool_release_page(): Unmap the page (if mapped) and account for it on inflight counters.
- page_pool_dev_alloc_pages(): Get a page from the page allocator or page_pool caches.
- page_pool_get_dma_addr(): Retrieve the stored DMA address.
- page_pool_get_dma_dir(): Retrieve the stored DMA direction.
- page_pool_put_page_bulk(): Tries to refill a number of pages into the ptr_ring cache holding ptr_ring producer lock. If the ptr_ring is full, page_pool_put_page_bulk() will release leftover pages to the page allocator. page_pool_put_page_bulk() is suitable to be run inside the driver NAPI tx completion loop for the XDP_REDIRECT use case. Please note the caller must not use data area after running page_pool_put_page_bulk(), as this function overwrites it.
- page_pool_get_stats(): Retrieve statistics about the page_pool. This API is only available if the kernel has been configured with `CONFIG_PAGE_POOL_STATS=y`. A pointer to a caller allocated `struct page_pool_stats` structure is passed to this API which is filled in. The caller can then report those stats to the user (perhaps via ethtool, debugfs, etc.). See below for an example usage of this API.

Stats API and structures

If the kernel is configured with `CONFIG_PAGE_POOL_STATS=y`, the API `page_pool_get_stats()` and structures described below are available. It takes a pointer to a `struct page_pool` and a pointer to a `struct page_pool_stats` allocated by the caller.

The API will fill in the provided `struct page_pool_stats` with statistics about the page_pool.

The stats structure has the following fields:

```
struct page_pool_stats {
    struct page_pool_alloc_stats alloc_stats;
    struct page_pool_recycle_stats recycle_stats;
};
```

The `struct page_pool_alloc_stats` has the following fields:

- `fast`: successful fast path allocations
- `slow`: slow path order-0 allocations
- `slow_high_order`: slow path high order allocations
- `empty`: ptr ring is empty, so a slow path allocation was forced.
- `refill`: an allocation which triggered a refill of the cache
- `waive`: pages obtained from the ptr ring that cannot be added to the cache due to a NUMA mismatch.

The `struct page_pool_recycle_stats` has the following fields:

- `cached`: recycling placed page in the page pool cache
- `cache_full`: page pool cache was full
- `ring`: page placed into the ptr ring
- `ring_full`: page released from page pool because the ptr ring was full
- `released_refcnt`: page released (and not recycled) because refcnt > 1

Coding examples

Registration

```
/* Page pool registration */
struct page_pool_params pp_params = { 0 };
struct xdp_rxq_info xdp_rxq;
int err;

pp_params.order = 0;
/* internal DMA mapping in page_pool */
pp_params.flags = PP_FLAG_DMA_MAP;
pp_params.pool_size = DESC_NUM;
pp_params.nid = NUMA_NO_NODE;
pp_params.dev = priv->dev;
pp_params.dma_dir = xdp_prog ? DMA_BIDIRECTIONAL : DMA_FROM_DEVICE;
page_pool = page_pool_create(&pp_params);

err = xdp_rxq_info_reg(&xdp_rxq, ndev, 0);
if (err)
    goto err_out;

err = xdp_rxq_info_reg_mem_model(&xdp_rxq, MEM_TYPE_PAGE_POOL, page_pool);
if (err)
```

```
    goto err_out;
```

NAPI poller

```
/* NAPI Rx poller */
enum dma_data_direction dma_dir;

dma_dir = page_pool_get_dma_dir(dring->page_pool);
while (done < budget) {
    if (some error)
        page_pool_recycle_direct(page_pool, page);
    if (packet_is_xdp) {
        if XDP_DROP:
            page_pool_recycle_direct(page_pool, page);
    } else (packet_is_skb) {
        page_pool_release_page(page_pool, page);
        new_page = page_pool_dev_alloc_pages(page_pool);
    }
}
```

Stats

```
#ifdef CONFIG_PAGE_POOL_STATS
/* retrieve stats */
struct page_pool_stats stats = { 0 };
if (page_pool_get_stats(page_pool, &stats)) {
        /* perhaps the driver reports statistics with ethool */
        ethtool_print_allocation_stats(&stats.alloc_stats);
        ethtool_print_recycle_stats(&stats.recycle_stats);
}
#endif
```

Driver unload

```
/* Driver unload */
page_pool_put_full_page(page_pool, page, false);
xdp_rxq_info_unreg(&xdp_rxq);
```