AGENDA

- Introduction
- Current status – The RAW ETH QP
- Receive Side Scaling
- L2 Tunneling stateless offloads
- Capturing
- Completion Queue – Support New Extensions
- User Mode Non-Privileged Access
- Conclusion
INTRODUCTION

- Telecom, Web 2.0, Cloud & FSI high-end applications increase network requirements
- Would like to reduce operating systems overhead
  - Data path direct User application to HW access APIs
  - Get high PPS rates, low latency, minimize cycle/byte and increased scalability
- Transpareently use standard TCP/UDP/IP protocols
  - No need for proprietary protocol designs
  - Use existing rich HW protocol offload support
  - Can interoperate with traditional OS TCP/IP stack
CURRENT STATUS – THE RAW ETH QP

- **Ibv_qp type:** RAW_ETH
- **Use mature verbs objects**
  - QP, CQ, MR
- **Pair of send and receive queues**
  - Send queue to transmit raw packets - No implicit headers
  - Receive queue is steered according to flows classification
- **Stateless Offloads Engine**
  - Currently csum offload is supported
  - And Interrupt moderation (CQ moderation)
- **Require privileged user**
  - CAP_NET_RAW
- Receive Side Scaling (RSS) technology enables spreading incoming traffic to multiple receive queues.
- Each receive queue is associated with a completion queue.
- Completion Queues (CQ) are bound to a CPU core:
  - CQ is associated with interrupt vector and thus with CPU.
  - For polling, user may run polling for each CQ from associated CPU.
  - In NUMA systems, CQ may be allocated on close memory to associated CPU.
- Spreading the receive queues to different CPU cores allows spreading receive workload of incoming traffic.
Classify first, distribute after

- **Begin with classification**
  - Using Steering (ibv_create_flow()) classify incoming traffic
  - Classification rules may be any of the packet L2/3/4 header attributes
    - e.g. TCP/UDP only traffic, IPv4 only traffic, ..
  - Classification result is transport object - QP

- **Continue with spreading**
  - Transport object (QPs) are responsible for spreading to the receive queues
  - QPs carry RSS spreading rules and receive queue indirection table

- **RQs are associated with CQ**
  - CQs are associated with CPU core

- **Different traffic types can be subject to different spreading**
Typically QPs (Queued Pairs) are created with 3 elements
- Transmit and receive Transport
- Receive Queue
  - Exception is QPs which are associated with SRQ
- Send Queue

Extend verbs to support separate allocation of the above 3 elements
- Transport – ibv_qp with no RQ or SQ
  - Ibv_qp_type of IBV_QPT_RAW_ETH
  - Next will be UD QP type
  - New QP attribute: ibv_rx_hash_conf
- Work Queue – ibv_wq
  - Can be of 2 types: IBV_RQ – Receive Queue and IBV_SQ
  - We’ll start with IBV_RQ definition
Work Queue (WQ) – Cont.

- **New object: Work Queue - `ibv_wq`**
- **Managed through following new calls:**
  - `ibv_wq *ibv_create_wq(ibv_wq_init_attr)`
  - `ibv_modify_wq(ibv_wq , ibv_wq_attr)`
  - `ibv_destory_wq(ibv_wq)`
  - `ibv_post_wq_recv(ibv_wq, ibv_recv_wr)`

- **Work Queues (ibv_wq) are associated with Completion Queue (ibv_cq)**
  - Multiple Work Queues may be mapped to same Completion Queue (many to one)

- **Work Queues of type Receive Queue (IBV_RQ) may share receive pull**
  - By associating many Work Queues to same Shared Receive Queue (the existing verbs `ibv_srq` object)

- **QP (ibv_qp) can be created without internal Send and Receive Queues and associated with external Work Queue (ibv_wq)**
- **QP can be associated with multiple Work Queues of type Receive Queue**
  - Through Receive Queue Indirection Table object

```c
struct ibv_wq {
    struct ibv_context   *context;
    void                 *wq_context;
    uint32_t             handle;
    struct ibv_pd        *pd;
    struct ibv_cq        *cq;
    /* SRQ handle if WQ is to be / associated with an SRQ, / otherwise NULL */
    struct ibv_srq        *srq;
    uint32_t             wq_num;
    enum ibv_wq_state    state;
    enum ibv_wq_type     wq_type;
    uint32_t             comp_mask;
};
```
RSS
WQ of Type RQ – State Diagram

- CREATE_RQ
- DESTROY_RQ
- MODIFY_RQ (RDY2RDY)
- MODIFY_RQ (RST2RDY)
- RST
- RDY
- ERR

Transition:
- SW
- SW/HW
New object: Receive Work Queue Indirection Table – ibv_rwq_ind_table
Managed through following new calls:
- ibv_wq_ind_tbl
- ibv_create_rwq_ind_table(ibv_rwq_ind_table_init_attr)
- ibv_modify_rwq_ind_table(ibv_rwq_ind_table)
- ibv_query_rwq_ind_table(ibv_rwq_ind_table, ibv_rwq_ind_table_attr)
- ibv_destroy_rwq_ind_table(ibv_rwq_ind_table)

QPs may be associated with an RQ Indirection Table
Multiple QPs may be associated with same RQ Indirection Table
“RSS” QP
- QP attributes (ibv_qp_attr) now include RSS hash configuration attributes (ibv_rx_hash_conf)
- QP is Stateless
- QP’s Send and Receive WQs parameters are invalid - QP has no internal work queues
- Use `ibv_post_wq_recv` instead of `ibv_post_recv`
- QP is connected to RQ Indirection Table

On Receive, traffic is steered to the QP according to existing steering API
- `Ibv_create_flow()`

Following, matching RQ is chosen according to QPs hash calculation

```c
struct ibv_rx_hash_conf {
    /* enum ibv_rx_hash_function */
    uint8_t  rx_hash_function;
    /* valid only for Toeplitz */
    uint8_t   *rx_hash_key;
    /* enum ibv_rx_hash_fields */
    uint64_t  rx_hash_fields_mask;
    struct ibv_rwq_ind_table  *rwq_ind_tbl;
};

/* RX Hash Function. */
enum ibv_rx_hash_function_flags {
    IBV_RX_HASH_FUNC_TOEPLTIZ = 1 << 0,
    IBV_RX_HASH_FUNC_XOR      = 1 << 1
};

/* Field represented by the flag will be used in RSS Hash calculation. */
enum ibv_rx_hash_fields {
    IBV_RX_HASH_SRC_IPV4       = 1 << 0,
    IBV_RX_HASH_DST_IPV4       = 1 << 1,
    IBV_RX_HASH_SRC_IPV6       = 1 << 2,
    IBV_RX_HASH_DST_IPV6       = 1 << 3,
    IBV_RX_HASH_SRC_PORT_TCP   = 1 << 4,
    IBV_RX_HASH_DST_PORT_TCP   = 1 << 5,
    IBV_RX_HASH_SRC_PORT_UDP   = 1 << 6,
    IBV_RX_HASH_DST_PORT_UDP   = 1 << 7
};
```
Verbs Steering Classifies the traffic  

IBV_QPT_RAW_PACKET QPs distributes traffic type between RQs/Cores

Verbs Flows

Verbs Flows

IBV_QPT_RAW_PACKET QPs with IBV_QP_INIT_ATTR_RX_HASH = 1

Enabled flags in rx_hash_fields_mask

TCP IPv4

rx_hash_function

Hash Value

QPs

Enabled flags in rx_hash_fields_mask

UDP IPv6

rx_hash_function

Hash Value

QPs

IBV_RWQ_IND_TBL  IBV_WQT_RQ  IBV_CQs  Cores

Core 1

Core 2

Core 3
IPoIB UD QP type

- “RSS” UD QP is connected to RQ Indirection Table
- RSS UD QP to continue to manage UD transport attributes: pkey, qkey checks…
- Single wire QPN for all getting to all the QPs Receive Queues

Transmit Side Scaling (TSS)

- As in RSS, QP is stateless, Send and Receive work queues attributes are invalidate
- Use ibv_post_wq_send instead of ibv_post_send
- For IPoIB UD QP:
  - Manage UD transport properties: pkey, qkey…
  - Use single source QPN in DETH wire protocol header for all Send WQ which is the “TSS” UD QP
- The same QP may be used for both “RSS” and “TSS” operations
Tunneling technologies like VXLAN, NVGRE, GENEVE were introduced for solving cloud scalability and security challenges.

Require extensions of traditional NIC stateless offloads:
- TX and RX inner headers checksum
  - `ibv_qp_attr` to control inner csum offload
  - `ibv_send_wr`, `ibv_wc` to request and report inner csum
- Inner TCP Segmentation and De-segmentation (LSO/LRO)
  - `ibv_send_wr` to support inner MSS settings
- Outer and inner Ethernet header VLAN insertion and stripping
  - `ibv_qp_attr` to control VLAN insert/strip
  - `ibv_send_wr` to indicate VLAN
  - `ibv_wc` to report strip VLAN
- Steering to QP according to outer and inner headers attributes
  - `ibv_create_flow(ibv_flow_attr)` to support inner headers
- Perform RSS based on inner or on outer header attributes
  - `ibv_qp_attr.ibv_rx_hash_conf` to support inner header attributes
- Inner packet parsing and reporting its properties in Completion Queue Entry (CQE)
  - `ibv_wc` to support inner headers extraction
**CAPTURING**

- **Support standard Capturing interfaces and solutions**
  - User mode Ethernet traffic (OS Bypass traffic) is capture-able like traditional TCP/IP stack traffic
  - For Linux: standard PF_PACKET RAW Socket libpcap support, i.e., utilities that use libpcap are supported: tcpdump, wireshark, …
  - Windows: Microsoft Message Analyzer (MMA)

- **Both TX and RX traffic**
- **Applicable for both ETH and RDMA traffic capturing**
CAPTURING
OS Bypass Capture App

- User mode OS bypass capturing application through Verbs API
  - Through ibv_create_flow() plus indicating sniffer flag
  - Classify requested captured traffic
  - Steer to QP. Can be “RSS” QP

**Sniffer Flow Table**

<table>
<thead>
<tr>
<th>Priority</th>
<th>Classification</th>
<th>Direction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MAC+VLAN</td>
<td>TX+RX</td>
<td>QP#10 SNIFF</td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ingress Traffic  Egress Traffic

RQ#0  RQ#1  RQ#2  RQ#3

QP#10
Extending Verbs Support for user mode Ethernet requires growing the Work Completion (ibv_wc)

More and more attributes are added to ibv_wc
- Completion time stamp
- Stripped VLAN
- Checksum and RSS hash result
- Tunneling inner headers information
- ...

Completion Queue polling (ibv_poll_cq(ibv_wc*)) is critical data path operation

Growing ibv_wc size will result in performance hit
- Increased cache misses
- Redundant extra copies of per vendor HW completion memory to SW completion memory (ibv_wc)

A single completion data for all use cases is obsolete
COMPLETION QUEUE (CQ)
New Extension Support - Verbs

Requirements
- Completion (CQE) attribute read according to application needs
- Per vendor optimizations for each read access
- Batch read of multiple Completions (CQE) followed by single read pointer update

ibv_cq is extended to include function pointers for completion handling
- Object oriented approach – no need to overpopulate general verbs function namespace
- Methods will support extracting each completion attribute
  - So each app can extract only relevant attributes
  - Each verbs provider (vendor) will build it’s extraction method
  - Additionally a single method will be provided for extracting mostly used attributes (opcode, status, ..)

Batch read support
- Ibv_begin_poll(ibv_cq*) – Grab CQ lock
- Ibv_next_poll(ibv_cq*) – Advance CQ read pointer
- Ibv_end_poll(ibv_cq*) – Update the provider with CQ read pointer (typically doorbell to HW)

```c
struct ibv_cq_ex {
    /* legacy ibv_cq fields */
    ibv_cq cq;
    int comp_mask;

    /* CQ management methods */
    int (*begin_poll_ex)(struct ibv_cq_ex *cq);
    int (*next_poll_ex)(struct ibv_cq_ex *cq);
    void (*end_poll_ex)(struct ibv_cq_ex *cq);

    /* Work Completion per attribute read methods */
    ibv_wc *(*ibv_read_wc)(struct ibv_cq_ex *cq);
    int (*read_result)(ibv_wc_opcode *opcode,
                       enum ibv_wc_status* status);
    uint64 (*read_time_stamp)(struct ibv_cq_ex *cq);
    field1_t (*read_field1)(struct ibv_cq_ex *cq);
    field2_t (*read_field2)(struct ibv_cq_ex *cq);
    ..}
```
- **RAW ETH QP allows app to build its own L2/3/4 headers**
  - Alike SOCK_RAW socket() type
- **Caller to `ibv_create_qp()` with QP type of RAW_ETH must have CAP_NET_RAW privileges**
  - Alike SOCK_RAW socket() type

- **Support non-privileged user - L2/3/4 headers must be controlled by OS**
- **Option I:**
  - Add new QP types: RAW_ETH_UDP, RAW_ETH_TCP
  - Use `ibv_ah` for RAW ETH QP
  - Add d.IP indication to `ibv_ah`
  - On `ibv_create_ah()` `ibv_core` will perform route and address resolution to determine source I/f and corresponding s.MAC, s.IP and d.MAC.
  - L2/L3 header info will be cached in `ibv_ah` and registered for updates in case neigh is updated
    - Perform period updates of kernel dst neigh aging timers
    - HW is configured to enforce headers checks
- **Option II:**
  - Stay with single QP Type: RAW_ETH
  - App still build L2/3/4 headers itself
  - HW is configured to enforce headers checks on allowed L2/3 addresses and L4 ports per QP
  - Allowed addresses, ports may be configured through `ibv_create_qp` and/or `ibv_create_flow()`

- **Continue supporting RAW access for privileged users**
CONCLUSION

- Verbs API infrastructure is a robust and efficient API
- Generic object model to expend to new I/O offloads
- Control and data path infrastructure
  - Use OS services for control path and allow bypass for data path
  - Can answer performance requirements for both high PPS, BW and low latency
- Extendable in backward and forward compatible manner through Verbs extensions

*Great platform to expand user mode Ethernet programming*
12th ANNUAL WORKSHOP 2016

THANK YOU

Tzahi Oved
Mellanox Technologies