libeventdev: Event driven programming model framework for DPDK

Jerin Jacob
jerin.jacob@cavium.com
 Agenda

● Event driven programming model concepts from data plane perspective
● Characteristics of HW based event manager devices
● libeventdev
● Example use case - Simple IPSec outbound processing
● Benefits of event driven programming model
Event driven programming model - Concepts

- Event is an *asynchronous* notification from HW/SW to CPU core
- Typical examples of events in dataplane are
  - Packets from ethernet device
  - Crypto work completion notification from Crypto HW
  - Timer expiry notification from Timer HW
  - CPU generates an event to notify another CPU (used in pipeline mode)
- Event driven programming is a programming paradigm in which flow of the program is determined by events
Event driven programming model - Concepts

Packet event, Timer expiry event and crypto work complete event are the typical HW generated events.

Queue 0..N stores the events.

Scheduler schedules an event to core.

Core process the event and enqueue to another downstream queue for further processing or send the event/packet to wire.

Core can also produce the SW event to notify another core for work completion.
Characteristics of HW based event device

- Millions of flow queues
- Events associated with a single flow queue can be scheduled on multiple CPUs for concurrent processing while maintaining the original event order
- Provides synchronization of the events without SW lock schemes
- Priority based scheduling to enable the QoS
- Event device may have 1 to N schedule groups
  - Each core can be a member of any subset of schedule groups
    - Each core decides which schedule group(s) it accepts the events from
    - Schedule groups provide a means to execute different functions on different cores
- Flow queues grouped into schedule groups
- Core to schedule group membership can be changed at runtime to support scaling and reduce the latency of critical work by assigning more cores at runtime
- Event scheduler is implemented in HW to save CPU cycles
libeventdev components

Scheduler

Core 0

Core 1

Core n

Packet event

Timer expiry

Crypto done ev

SW event

enqueue(grp_id, flow_queue_id, schedule_sync, event_type, event) = schedule()

priority x

priority y

priority z

Core 0's Sched Group bitmask: 100011

Group 0

Group 1

Group n

Core 1's Sched Group bitmask: 000001

Group 0

Group n

Each core has group-mask to capture, the list of schedule groups participate in schedule()
libeventdev - flow

- Event driver registers with libeventdev subsystem and subsystem provide a unique device id
- Application get the device capabilities with rte_eventdev_info_get(dev_id), like
  - The number of schedule groups
  - The number of flow queues in a schedule group
- Application configures the event device and each schedule groups in the event device, like
  - The number of schedule groups and the flow queues are required
  - Priority of each schedule group and list of l-cores associated with it
  - Connect schedule groups with other HW event producers in the system like ethdev and crypto etc
- In fastpath,
  - HW/SW enqueues the events to flow queues associated with schedule groups
  - Core gets the event through scheduler by invoking rte_event_scheduler() from lcore
  - Core process the event and enqueue to another downstream queue for further processing or send the event/packet to wire if it is the last stage of the processing
  - rte_event_scheduler() schedules the event based on
    - selection of the schedule group
      - The caller l-core membership in the schedule group
      - Schedule group priority relative to other schedule groups.
    - selection of the flow queue and the event inside the schedule group
      - Scheduler sync method associated with the flow queue(ATOMIC vs ORDERED/PARALLEL)
Schedule sync methods (How events are Synchronized)

- **PARALLEL**
  - Events from a parallel flow queue can be scheduled to multiple cores for concurrent processing
  - Ingress order is *not* maintained

- **ATOMIC**
  - Events from an atomic flow queue can schedule only to a *single core* at a time
  - Enable critical section in packet processing like sequence number update etc
  - Ingress order is *maintained* as outstanding is always one at a time

- **ORDERED**
  - Events from the ordered flow queue can be scheduled to multiple cores for concurrent processing
  - Ingress order is *maintained*
  - Enable high *single flow* throughput
ORDERED flow queue for ingress ordering

The source ORDERED flow queue’s ingress order shall be maintained when events are enqueued to any downstream flow queue.
Use case (Simple IPSec Outbound processing)

PHASE1: POLICY/SA, ROUTE Lookup In parallel (ORDERED)

PHASE2: SEQ Number update per SA (ATOMIC)

PHASE3: HW assisted IPSec crypto

PHASE4: Core sends encrypted pks to Tx port queues (ATOMIC)

Packets enqueued into one of up to 1M flow queues based on a classification criterion (e.g., 5 tuple hash)

Core issues IPSec crypto request to HW

Crypto HW sends the crypto work completion event to notify the core.

PHASE1 generates a unique SA based on input packet and SA tables. Each SA flow will be processed in parallel. Core enqueues on ATOMIC flow queue for critical section processing per SA

Crypto HW processes the crypto operations in background

Core issues IPSec crypto request to HW
while(1) {
    event = rte_event_schedule();
    process the specific phase
    call different enqueue() to send to
    - atomic flow queue
    - crypto HW engine queue
    - TX port queue
}

Per SA, Core enqueues on ATOMIC flow queue for critical section phase of the flow

On completion of crypto work, HW generates the crypto work completion notification

RX pkt HW enqueues one of millions flow to ORDERED flow queues
#define APP_STATE_SEQ_UPDATE 0

```c
on each lcore
{
  struct rte_event ev;
  uint32_t flow_queue_id_mask = rte_eventdev_flow_queue_id_mask(eventdev);

  while (1) {
    ret = rte_event_schedule(eventdev, &ev, true);
    if (!ret)
      continue;

    /* packets from HW rx ports proceed parallely per flow(ORDERED)*/
    if (ev.event_type == RTE_EVENT_TYPE_ETHDEV) {
      sa = outbound_sa_lookup(ev.mbuf);
      /* move to next phase (atomic seq number update per sa) */
      ev.flow_queue_id = sa & flow_queue_id_mask;
      ev.sched_sync = RTE_SCHED_SYNC_ATOMIC;
      ev.sub_event_id = APP_STATE_SEQ_UPDATE;
      rte_event_enqueue(eventdev, ev);
    } else if (ev.event_type == RTE_EVENT_TYPE_LCORE && ev.sub_event_id == APP_STATE_SEQ_UPDATE) {
      sa = ev.flow_queue_id;
      /* do critical section work per sa */
      do_critical_section_work(sa);
      ev.flow_queue_id = tx_port;
      ev.sub_event_id = tx_queue_id;
      ev.sched_sync = RTE_SCHED_SYNC_ATOMIC;
      rte_cryptodev_event_enqueue(cryptodev, ev.mbuf, eventdev, ev);
    } else if (ev.event_type == RTE_EVENT_TYPE_CRYPTODEV) {
      tx_port = ev.flow_queue_id;
      tx_queue_id = ev.sub_evend_id;
      send the packet to tx port/queue
    }
  }
```

/* Issue the crypto request and generate the following on crypto work completion */
```c
} else if((ev.event_type == RTE_EVENT_TYPE_CRYPTODEV)
  tx_port = ev.flow_queue_id;
  tx_queue_id = ev.sub_evend_id;
  send the packet to tx port/queue
```

/* packets from HW rx ports proceed parallely per flow(ORDERED)*/
if (ev.event_type == RTE_EVENT_TYPE_ETHDEV) {
  sa = outbound_sa_lookup(ev.mbuf);
  modify the packet per SA attributes
  find the tx port and tx queue from routing table

  /* move to next phase (atomic seq number update per sa) */
  ev.flow_queue_id = sa & flow_queue_id_mask;
  ev.sched_sync = RTE_SCHED_SYNC_ATOMIC;
  ev.sub_event_id = APP_STATE_SEQ_UPDATE;
  rte_event_enqueue(eventdev, ev);
} else if (ev.event_type == RTE_EVENT_TYPE_LCORE &&& ev.sub_event_id == APP_STATE_SEQ_UPDATE) {
  sa = ev.flow_queue_id;
  /* do critical section work per sa */
  do_critical_section_work(sa);

  /* Issue the crypto request and generate the following on crypto work completion */
  ev.flow_queue_id = tx_port;
  ev.sub_event_id = tx_queue_id;
  ev.sched_sync = RTE_SCHED_SYNC_ATOMIC;
  rte_cryptodev_event_enqueue(cryptodev, ev.mbuf, eventdev, ev);
}
Benefits of event driven programming model

- Enable high *single flow* throughput with ORDERED schedule sync method
- The processing stages are not bound to specific cores. It provides better load-balancing and scaling capabilities than traditional pipelining.
- Prioritize: Guarantee lcores work on the highest priority event available
- Support *asynchronous* operations which allow the cores to stay busy while hardware manages requests.
- Remove the static mappings between *core* to *port/rx queue*
- Scaling from 1 to N flows are easy as its not bound to specific cores
Backup slides
API Requirements

• APIs similar to existing ethernet and crypto API framework for
  ○ Device creation, device Identification and device configuration
• Enumerate libeventdev resources as numbers(0..N) to
  ○ Avoid ABI issues with handles
  ○ event device may have million flow queues so it's not practical to have handles for each flow queue and its associated name based lookup in multiprocess case
• Avoid struct mbuf changes
• APIs to
  ○ Enumerate eventdev driver capabilities and resources
  ○ Enqueue events from I-core
  ○ Schedule events
  ○ Synchronize events
  ○ Maintain ingress order of the events
API - Slow path

- APIs similar to existing ethernet and crypto API framework for
  
  - **Device creation**: Physical event devices are discovered during the PCI probe/enumeration of the EAL function which is executed at DPDK initialization, based on their PCI device identifier, each unique PCI BDF (bus/bridge, device, function)

  - **Device Identification**: A unique device index used to designate the event device in all functions exported by the eventdev API.

  - **Device Capability discovery**
    - **rte_eventdev_info_get()**: To get the global resources like number of schedule groups and number of flow queues per schedule group etc of the event device

  - **Device configuration**
    - **rte_eventdev_configure()**: configures the number of schedule groups and the number of flow queues on the schedule groups
    - **rte_eventdev_sched_group_setup()**: configures schedule group specific configuration like priority and the list of l-core has membership in the schedule group

  - **Device state change**: **rte_eventdev_start()**/**stop()**/**close()** like ethdev device
API - Fast path

- **bool rte_event_schedule**(uint8_t dev_id, struct rte_event *ev, bool wait);
  - Schedule an event to the caller l-core from a specific schedule group of event device designated by its dev_id
- **bool rte_event_schedule_from_group**(uint8_t dev_id, uint8_t group_id, struct rte_event *ev, bool wait);
  - Like rte_event_schedule(), but schedule group provided as argument
- **void rte_event_schedule_release**(uint8_t dev_id);
  - Release the current scheduler synchronization context associated with the scheduler dispatched event
- **int rte_event_schedule_group_[join/leave]**(uint8_t dev_id, uint8_t group_id);
  - Leave/Joins the caller l-core from/to a schedule group
- **bool rte_event_schedule_ctxt_update**(uint8_t dev_id, uint32_t flow_queue_id, uint8_t sched_sync, uint8_t sub_event_type, bool wait);
  - rte_event_schedule_ctxt_update() can be used to support run-to-completion model where the application requires the current *event* to stay on the same l-core as it moves through the series of processing stages, provided the event type is RTE_EVENT_TYPE_LCORE
Run-to-completion model support

- `rte_event_schedule_ctxt_update()` can be used to support run-to-completion model where the application requires the current *event* to stay on same l-core as it moves through the series of processing stages, provided the event type is `RTE_EVENT_TYPE_LCORE` (l-core to l-core communication).

- For example in the previous use case, the ATOMIC sequence number update per SA can be achieved like below:

```c
/* move to next phase (atomic seq number update per sa) */
ev.flow_queue_id = sa & flow_queue_id_mask;
ev.sched_sync = RTE_SCHED_SYNC_ATOMIC;
ev.sub_event_id = APP_STATE_SEQ_UPDATE;
rte_event_enqueue(eventdev, ev);
}
else if (ev.event_type == RTE_EVENT_TYPE_LCORE && ev.sub_event_id == APP_STATE_SEQ_UPDATE) {
    sa = ev.flow_queue_id;
    /* do critical section work per sa */
    do_critical_section_work(sa);
    /* move to next phase (atomic seq number update per sa) */
    rte_event_schedule_ctxt_update(eventdev, sa & flow_queue_id_mask, RTE_SCHED_SYNC_ATOMIC, APP_STATE_SEQ_UPDATE, true);
    /* do critical section work per sa */
    do_critical_section_work(sa);

Scheduler context update is costly operation, by splitting it as two functions(`rte_event_schedule_ctxt_update()` and `rte_event_schedule_ctxt_wait()`) allows application to overlap the context switch latency with other profitable work.
Future work

- Integrate the event device with ethernet, crypto and timer subsystems in DPDK
  - Ethdev/event device integration is possible by extending new 6WIND’s ingress classification specification where a new action type can establish ethdev’s port to eventdev’s schedule group connection
  - Cryptodev needs some change at configuration stage to set crypto work complete event delivery mechanism
  - Spec out timerdev for PCI based timer event devices (timer event devices generates timer expiry event vs callback in the existing SW based timer scheme)
  - Event driven model operates on a single event at a time. Need to create a helper API to make it burst in nature for the final enqueues to different HW block like ethdev tx-queue