User Perspectives On Trying To Use DPDK For Accelerating Networking In End-System Applications

Sowmini Varadhan (sowmini.varadhan@oracle.com)

Agenda

- Brief description of the types of networking paradigms typically encountered in database/cluster applications
- Some experiments in trying to use DPDK in these paradigms.
- Latency measurements, software-engineering considerations
- Conclusions from these experiments

Typical Oracle/DB Problem Space

- Primarily request/response Transactions
- Multithreaded applications, each application typically handling multiple descriptors.
- Networking: typically datagram sockets, using BSD socket based APIs
  - UDP sockets, or,
  - RDS (Reliable Datagram Service) sockets

DB Application Network/Socket Mode

- RDS: Reliable Datagram Service
- Application payload is encapsulated in an RDS header and handed off to some transport that guarantees reliable, ordered delivery
- Transport can be InfiniBand (bypasses TCP/IP stack), or TCP/IP/Ethernet.
- UDP based model is similar, but application has to do extra work to ensure reliable/ordered delivery.

Can We Use DPDK To Accelerate This?

- Some services are CPU bound, latency sensitive
- DPDK allows us to read packets directly from the driver (like IB) and we already have some infra to take care of guaranteed/ordered delivery, so evaluate if/how much latency reduction we can get from DPDK
  - If necessary, we can use our own custom ULP encapsulation over L4.

Using DPDK and KNI?

- DPDK inserts user-space PMDs, so all frames from that NIC are diverted to uspace
- We only want a subset of flows, we dont want to implement every IETF/IEEE protocol in our experiment, so try to use KNI
- Register a receive side callback with the user-space poll-mode driver to filter out “interesting” flows.
  - Interesting flows will be processed by DB software
  - Rest (NFS, SMTP, IP fragments, Routing protocol packets...) is sent to Linux stack via KNI's “vEthX”.

Observations about this approach

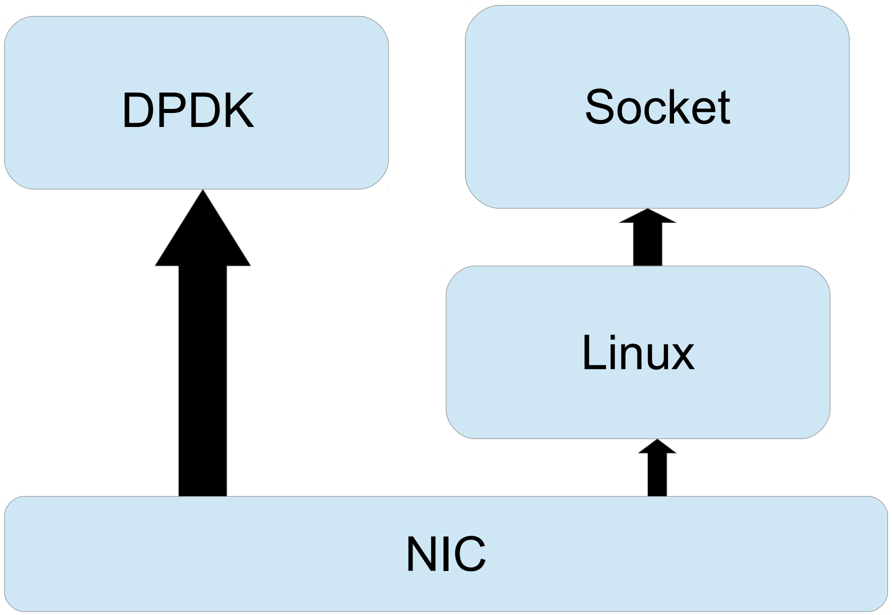
- It slows down linux TCP/IP stack
  - How much penalty to the TCP/IP stack?
- Instrument three different types of RTT and compare the numbers:
  - Direct path to DPDK (igb_uio ↔ PMD ↔ kni_rx_cb)
  - DPDK ↔ FIFO/shmem ↔ linux application
  - PF_PACKET ↔ linux application
- Experiment details: 64 byte sized packet with custom ethertype (i.e., flow selection by ethertype). Application on the SUT just reflects packet back by swapping dmac, smac.

Avoiding the penalty for the Linux stack

- Latency estimates:
  - Direct path to DPDK- approx (90 µs)
  - DPDK → FIFO/shmem → linux app (2000 µs)
  - PF_PACKET → linux application (150-200 µs)
- High penalty for apps using the linux stack, e.g., NFS, mail, ssh etc!
- Common practice: traffic bifurcation using SR-IOV
  - http://rhelblog.redhat.com/2015/10/02/
  - https://blog.cloudflare.com/kernel-bypass/

SR-IOV based traffic bifurcation

- Create a VF for the PCIe bus
- Use ethtool to set up a traffic filter to pull out "interesting" packets on the VF
- DPDK PMD drivers work with the VF, no penalty for linux stack
- Non-trivial routing, forwarding, ARP, egress adjacencies still needs special config



Additional Drawbacks..

- Dealing with IP fragmentation/re-assembly:
  - Even if we define the flow by the L4 4-tuple, we have to deal with IP fragments
  - This is slow path, we can let the native kernel stack implementation sort this out for us, but..
  - Application can now get packets either via DPDK path, or from native kernel stack
- In general, the application is reading from multiple I/O descriptors
  - Network packets from the wire,
  - Disk I/O..
- This is typically done using a select/poll/epoll loop.

Classic Server Side Paradigm:

```
while (1) {
   select(nfds, fd_set, ..);
   /* fd_set may have a mix of TCP, UDP, RDS
    * sockets. After select(), multiple fd's
    * in the set may be ready
    */
   if (incoming client connection request)
      accept() and add new socket to fd_set
   else { /* incoming data */
      Read request;
      Send response;
   }
}
```

Mapping this to DPDK?

- Recommendation was to try to use Rx Interrupt Mechanism: have and wait on a thread per fd,thread will be woken up when packets arrive.
- Major application rewrite needed to adapt to this?
- DPDK **library** would be placing constraints on the application's threading model
  - Signal delivery issues for single-threaded applications.
- examples/netmap_compat: gives a poll()-ish “fake” file desc with several critical restrictions.
  - Perf comparisons of netmap_compat vs native netmap ongoing.

CPU utilization in busy-poll mode

- DPDK kni example run (uses syntax supplied with DPDK package)

  # kni -c 0xf0 -n 4 – -P -p 0x1 –config “(0, 4, 5)”

- See DPDK documentation for details of what each arg in this incantation means..
- The effect of this set of arguments is that the poll-mode driver will use CPU 4 for Rx, CPU 5 for Tx.
  - CPUs 4 and 5 will be reported 0% idle on mpstat, even when there is no traffic flowing.
- 100% polling has problems: CPU power limits, PCI bus overhead

Conclusions

- DPDK may be ideal for "hot-potato" forwarding use-cases like ovs, where there are external protocols to set up the forwarding/switching rules, and DPDK is only used to accelerate the core forwarding engine.
- For End-System use-cases,
  - APIs matter. Ease of programmability is important.
  - Need to find an efficient way to co-exist with the existing kernel stack as the fallback for "uninteresting" (to the application) flows and network protocols.
  - Control plane considerations: Observability, Configuration

Backup Slides

Other user pleasers

- Existing “ethtool” application does not give visibility into offloads, detailed driver state..
- Better examples showing how to use h/w offload features

References

- "DPDK Performance : How not to just do a demo with DPDK" from Netdev 0.1
  - http://www.slideshare.net/shemminger/dpdk-performance

Managing file descriptors with DPDK

- examples/netmap_compat: gives a poll()-ish “fake” file desc
  - Cannot poll for > 0 or infinite (-1) timeout. Cannot add this to an fd_set that has other I/O descriptors
  - Multiple applications cannot open netmap sockets that listen on the same port e.g., cannot run two instances of the following from netmap_compat:

```
# ./build/bridge -c 0xf0 -n 4 - -i 0
  ## See next slide for details
```

  - Multiple threads in an app can't create netmap fd's on the same port (netmap_regif() fails)

Barriers to running multiple copies of netmap_compat:

- Hugepage allocation will fail: get_num_hugepages() hogs up all available free_pages.
  - *set num_pages at run time using gdb.*
- Each instance of the example tries to lock /var/run/.rte_config
  - *reset default_config_dir in eal_runtime_config_path() at run time using gdb.*
- Second instance of netmap_compat/bridge then makes the first instance SEGV.