Evaluating and improving kernel stack performance for datagram sockets from the perspective of RDBMS applications

Sowmini Varadhan(sowmini.varadhan@oracle.com)
Tushar Dave(tushar.n.dave@oracle.com)

Agenda

- What types of problems are we trying to solve?
- Possible solutions considered
- Benchmarks used in the RDBMS env
  - General networking microbenchmarks
  - Cluster IPC library benchmarks
- Some results from these benchmarks for UDP, PF_PACKET, RDS-TCP
- Next steps..

What types of problems are we trying to solve?

Two types of use-cases for reducing latency:

- Cluster applications that are CPU-bound and can benefit from reduced network latency
  - Specific UDP flows that can be identified by a 4-tuple
  - Request-response, transaction based. Request size: 512 bytes; Response size: 8192 bytes
- Extract Transform Load (ETL): input comes in JSON, CSV (comma-separated values) etc formats to Compute Node. Needs to be transformed to RDBMS format and stored to disk
  - Input comes in at a very high rate (e.g., from Trading) and needs to be processed as efficiently as possible
  - https://docs.oracle.com/database/121/DWHSG/ettover.htm#DWHSG011

Benchmarking with the Distributed Lock Management Server (LMS)

- Evaluate with Lock Management Server (LMS)
- LMS: Distributed request-response environment
- “Server” is a set of processes in the cluster that is the lock manager.
- Each client picks a port# from a port-range and sends a UDP request to a server at that port
  - Port-range is dynamically determined. Currently getting a well-balanced hash, even without REUSEPORT
- Client is blocked until response comes back.
- Client has to process response before it can send the next request

I/O patterns in the LMS environment

- Server is the bottleneck in this environment
  - Server side computation are CPU bound
  - Client is blocked until response is received.
- Client has to process the response before it can generate the next request
  - Input tends to be bursty
- Server side Rx batching is easy to achieve- server keeps reading input until it either runs out of buffer space or runs out of input
- Tx side batching is trickier: client is blocked until server sends response back, so excessive batching at the server will make input even more bursty.

Bottlenecks in the LMS environment

- System calls: each time the server has to read/write a packet, the system calls to recvmsg/sendmsg are an overhead
- Control over batch-size: each time the server runs out of input, if it has to fall back to poll(), the resulting context switch is expensive.
  - Control over optimal batch size for some packet Rx rate
- The expectation is that PF_PACKET/TPACKET_V* will help in above two areas

Requirements from latency accelarating solutions

- Need a select()able socket.
  - DB applications get I/O from multiple sources (disk, fs, network, etc). So network I/O **must** be on a socket that can be added to a select()/poll()/epoll() fd set.
- Accelerating latency of a subset of UDP flows **must not** be at the cost of regressed latency for other network packets
  - Solution must co-exist harmoniously with the existing linux kernel stack for other network protocols.
- Solution should not be intrusive.
  - Replacing socket creation, read and write routines is ok, but major revamp of application threading model is not acceptable.
- Support common POSIX/socket options like SNDBUF, RCVBUF, MSG_PEEK, TIMESTAMP..

Solutions considered (and discarded)

- DPDK
  - No select()able socket, not POSIX, radically different threading model.
  - Does not co-exist harmoniously with kernel stack: KNI huge latency burden for flows punted to linux stack; SRIOV-based solutions dont have a good way of correctly keeping in sync with linux control plane to figure out the egress packet dst headers.
- Netmap
  - Preliminary micro-benchmarking did not show significant perf benefit over PF_PACKET
  - exposes a lot of the driver APIs to user-space
  - Host-rings solution to share packets with the kernel stack was found to be problematic in our experiments
- PF_RING
  - Another way of doing PF_PACKET/PACKET_V2?

Solutions evaluated

- Evaluate
  - UDP with sendmsg/recvmsg
  - UDP with recvmmsg
  - PF_PACKET with TPACKET_V2, TPACKET_V3
- Expectation is that PF_PACKET with TPACKET_V* will help by reducing system-calls and improved control over the batching
- Benchmarks:
  - General networking benchmarks (netperf)
  - Convert Cluster IPC libraries (IPCLW) to use these mechanisms and evaluate using ipclw microbenchmarks.
  - Run “CRTEST” suite and evalute the ipclw library

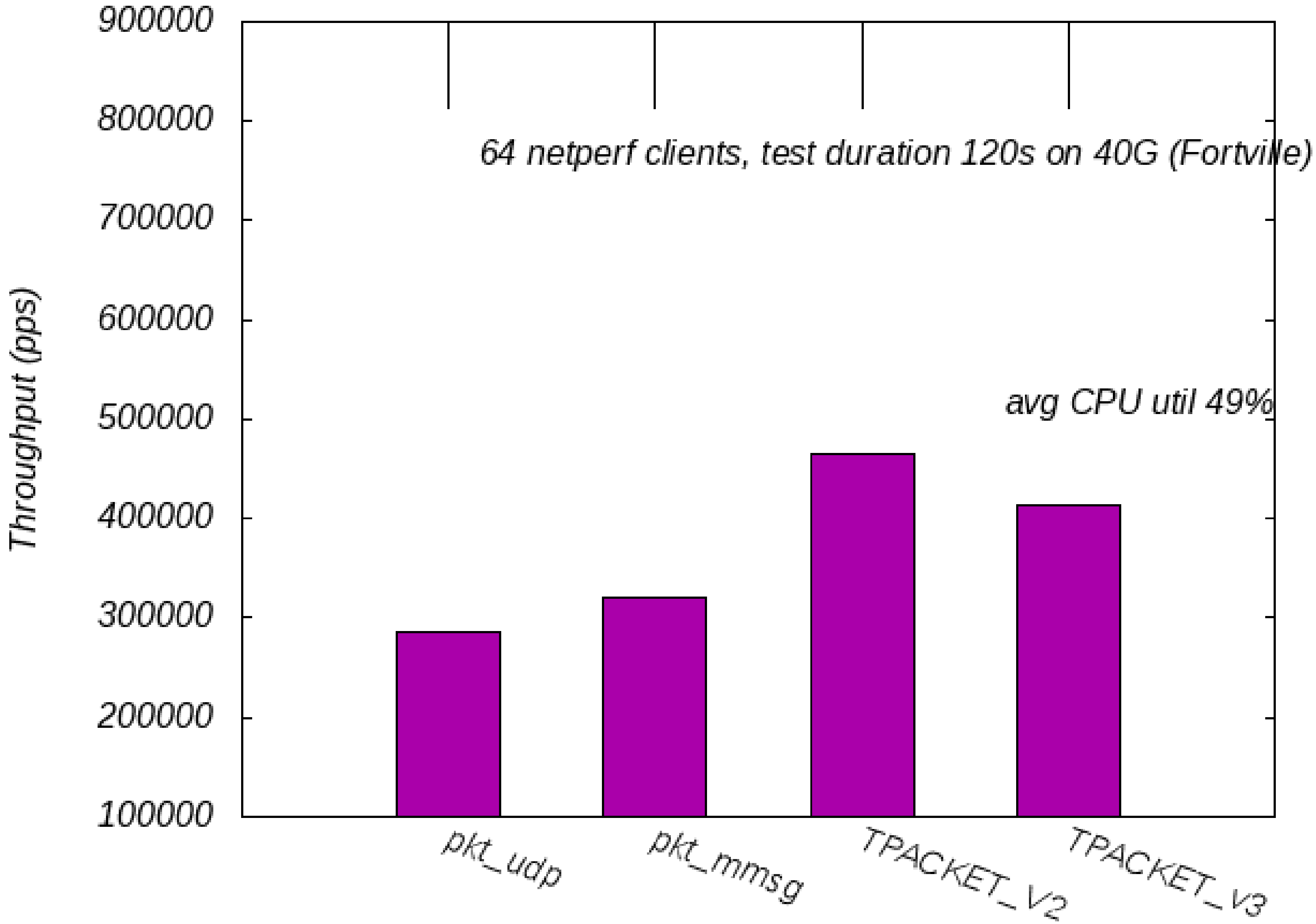
General networking microbenchmarks

- Standard netperf UDP_RR was used as the client for this evaluation, with parameters: req size 512, resp size 1024 (8K experiments use Jumbo frames on NIC, at the current time)
  - Netperf run with -N arg (nocontrol)
  - 64 netperf clients started in parallel
  - Flow hashing using address, port
- Application running the solution under evaluation listens in userspace, and sends back the UDP responses to netperf. Solutions evaluated were
  - UDP sockets with recvmsg()
  - UDP sockets with recvmmsg()
  - PF_PACKET with TPACKET_v2 and TPACKET_v3

Server side app details

- “pkt_udp”: simplistic batching; keep looping in {recvfrom(); sendto();} while there are packets to eat, else fall back to poll()
- “pkt_mmsg”: infinite timeout, vlen (batchsize) = 64
- “pkt_mmap”, single-threaded server test
  - TPACKET_V2, 16 frames-per-block, 2048 byte frames
  - TPACKET_v3, tmo = 10 ms, optimal sized frames/block for best perf and CPU util
- NIC was set up to do RSS using addr, port as rx-hash (i.e., sdfn setting for ethtool)

Netperf : single-threaded throughput



TPACKET_V3 batching behavior

Frames per block(fpb)	Tput (pps)	CPU-idle (%)
16	449543	0.94
32	419282	35
64	11639	99

- Gives more control over rx batching with Frame-per-Block(fpb) and timeout(TMO)
- Server thread is woken up either after block is full of requests or after timeout (to avoid infinite sleep)

64 clients sending requests and 1 server thread processing....

- fpb=16, server block easily become full, once woken up server thread remain woken up because it always have request to process, causes CPUs to be 99% busy.
- fpb=64, takes a little while for block to become full, server thread remains asleep and woken up when block is full; noticeable tput reduction and CPUs are almost idle.
- fpb=32, gives a good balance between Tput and CPU utilization.

Q: Can fpb dynamically managed depending on burst of client requests?

CPU utilization vs number of polls/sec

- The CPU utilization and the rate (per second) of the number of fallbacks to poll() was instrumented
- For UDP, recvmmsg() and TPACKET_V2
  - The CPU is kept 100% busy
  - At steady state (when all the netperf clients are up and running) we never fall back to poll()- there is always Rx input to be handled
- With TPACKET_V3, the application has more control over the batch size, and the timeout (for →sk_data_ready wakeup)
  - For max throughput, we can keep cpu at 100% busy
  - But, by adjusting frames/block and timeout, we can better the recvmmsg perf and keep CPU at 50% idle. Average polls/sec in this scenario is about 13.7.
  - When the clients are not able to fill the Rx pipe, server has fine-grained control over batching parameters

Converting IP Clusterware library (ipclw) to use PF_PACKET (in progress)

- the clusterware software is a library that is linked in by many applications; ongoing work to convert this to use PF_PACKET/TPACKET_V*
- Ether and IP header have to be supplied by the application:
  - need a separate thread that reads/writes on netlink sockets to keep in sync with kernel control plane
- Currently using Jumbo frames to send 8K responses, but this does not work when the dst is not directly connected
  - Either need IP frag management in user space or need UFO
- Currently skipping UDP checksum. In Production, we would need to offload UDP cheksum with PF_PACKET

Using CRTEST suite for verifying IPCLW

- A series of Cluster atomic benchmark tests for evaluating IPC performance. Simulates a typical RDBMS workload.
- Transfer data blocks over the cluster interconnect.
- Uses the IPCLW library for IPC, with various transports e.g., RDS-TCP, UDP, RDS-IB
- The LMS server node will have its buffer cache warmed up with “XCUR” buffers for all blocks in the test object.
  - XCUR == Exclusive Current. Only the instance that holds this Exclusive lock can change the block
- The client node will SELECT single blocks: read-only request that causes the instance holding the XCUR lock to make a “Consistent Read” (CR) copy that is shipped to the instance requesting the lock.

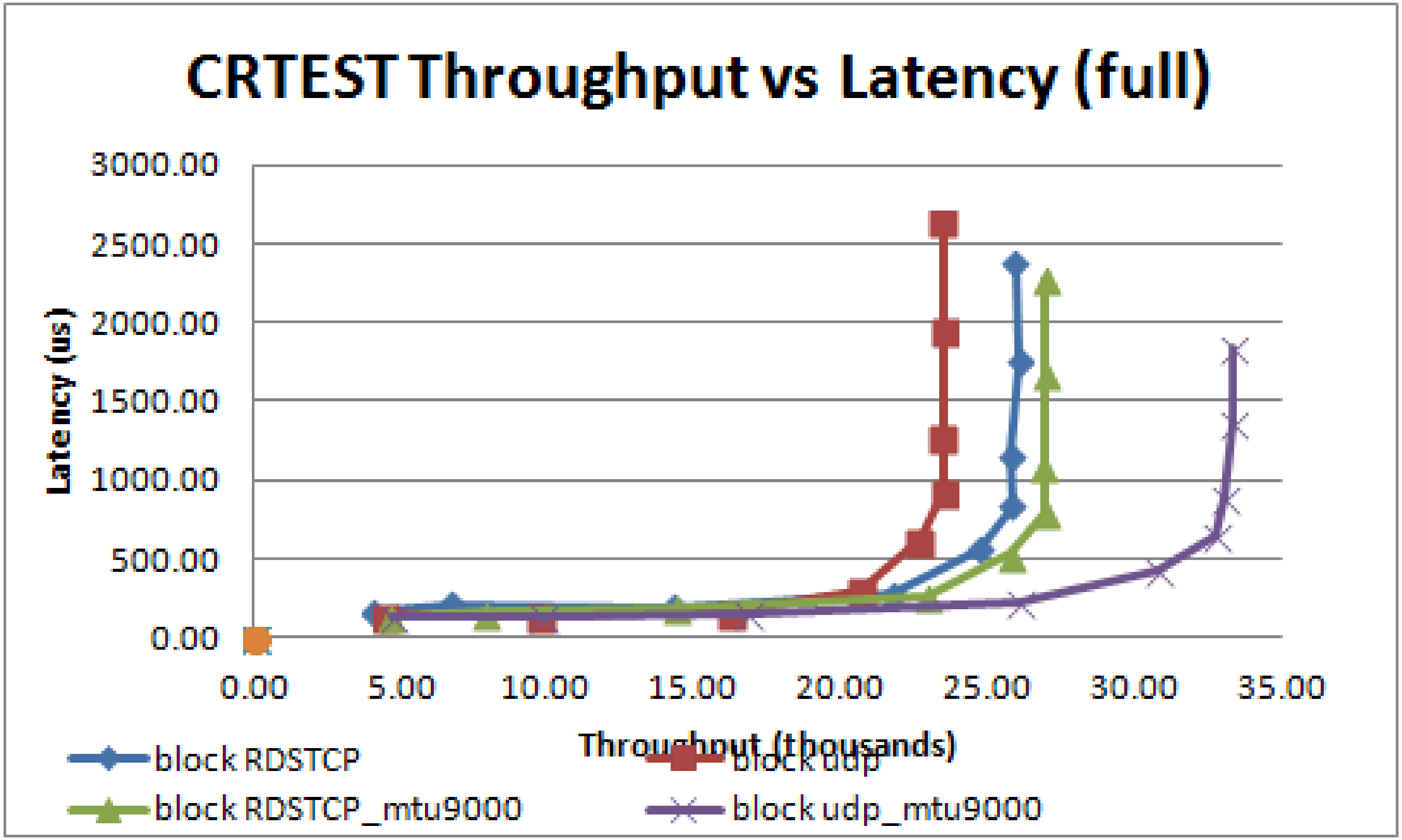
Handling large UDP packets

- CPU utilization is a bottleneck: now that the application can process packets faster, it's keep the CPU util at 100%, so any stack latency reduction is desirable
- If large UDP packets have to be broken down to a smaller MTU, something needs to do the IP frag/reassembly
  - UDP fragmentation offload to the NIC

CRTEST: test parameters

- Tested with nclients: {1, 2, 4, 8, 16, 24, 32, 48, 64}
- Both (single-path) RDS-TCP and UDP transports were tested
- For each value of nclients, instrument throughput and latency
- Objective:
  - Compare perf of RDS-TCP and UDP
  - Use Jumbo frames as an emulation of UDP fragmentation offload (UFO) to see if/how much it helps

CRTEST results



Thanks to yasuo.hirao@oracle.com for generating CRTEST data

CRTEST analysis

- The “wall” is a result of the server-side bottleneck.
  - As we increase the number of clients, there is a single server processing requests and sending responses. At the “wall”, we’ve hit the server side latency bottleneck: adding more clients does not increase throughput, but client requests spend more time on queue, so increase latency
- Why is the RDS-TCP “wall” to the right of UDP?
  - RDS-TCP has a single engine for tracking reliable, ordered, guaranteed delivery in the kernel
  - UDP runs multiple copies of seq/ack tracking engines in user-space. Thus it uses up more CPU for these engines, plus it is more vulnerable to scheduling delays in uspace (causing ACK timeout, unnecessary retransmits etc).

CRTEST and Jumbo frames

- Both throughput and latency improve significantly for UDP when going from 1500 → Jumbo MTU!
  - Latency: 2600 μs → 1800 μs
  - Throughput: 22K → 25K blocks/s (8192 bytes/block)
- Why doesn't TCP show the same jump in perf improvement?

Benefits of Jumbo for UDP vs TCP

- UDP protocol layer is stateless (esp in comparison with TCP)- most of the heavy lifting is done in the IP layer, around IP fragmentation/reassembly
  - Enabling Jumbo takes away a large part of that overhead- much better CPU utilization and throughput
- TCP already has TSO enabled, so it is able to send down large data packets to the driver
- Even with TSO, TCP has to manage a lot of protocol state, so the benefit of Jumbo is less than the equivalent for UDP
- Moral: UFO could vastly benefit many UDP based protocols!

Microbenchmarking vs production: lessons learned

- System tuning has to be done with caution: cannot favor of one flow/protocol/packet-size if it hurts some other feature/flow
  - e.g., cannot disable iommu, ethernet flow-control, tweak sysctl tunables in favor of specific TCP/UDP socket flavors
  - Cannot tune ethtool with sdfn
  - tcpdump and other packet consumers must continue to work - need to co-exist with host-stack
- Cannot rely on Jumbo for handling large packet sizes
  - Frag/reassembly challenges must be confronted.
- Cannot really fully exploit the benefits of shared memory when shimming things through a library

Exploiting zerocopy/shmem

- Even though TPACKET_V* allows the application to use shared memory, end up having to memcpy the packet to/from a user buffer in the library
- Reason: application calls some library function for read/write, and provides a buffer. Library has no control over when that buffer will eventually be released back to the kernel.
- One area where we can shave off a bcopy is by DMA-ing directly into the shmem buffer (avoid the sk_buff copy on Rx side)..
- Others?

Ongoing work

- Working on converting ipclw libraries to use PF_PACKET/TPACKET_V2, TPACKET_V3
- More NIC support for UFO
  - Can send down arbitrarily large frames to driver
  - Will give much better CPU utilization for many protocols that encaps in UDP (more and more of these showing up!)
  - Challenge may be UDP checksum of very large packets?
- Extend some of the TPACKET ideas for other socket types like RDS?