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

Sowmini Varadhan, Tushar Dave
Oracle Corporation,
Redwood City, CA
{sowmini.varadhan, tushar.n.dave}@oracle.com

Abstract

Applications implementing Relational Database Management Services (RDBMS Applications) use stateless datagram sockets such as RDS[9] and UDP[8]. These workloads are typically highly CPU-bound and sensitive to network latency, so any performance enhancements to these networking paths is attractive for RDBMS workloads. We share some findings from our benchmarking experiments using the Linux kernel for datagram based networking sockets in RDBMS applications and discuss potentials for improving in-stack performance using socket types such as PF_PACKET. Socket types such as PF_PACKET allow some benefits such as shared memory between user and kernel, and offer the benefits of a streamlined data-path with minimal data-copy. However, incorporating these methods into user-space software libraries for Database applications must satisfy existing API constraints which we describe in this paper.

As part of this effort we also gained some insights into generic performance analysis methods and the restrictions imposed by real-world workloads that are running multiple workloads with varying packet processing profiles. We share some of the insights gained in this paper.

Keywords

RDBMS, PF_PACKET, UDP, Benchmarking

Introduction

Relational Database Management Service environments are composed of a mix of applications, many of which involve transaction based processing over the network. The services offered by these applications tend to be highly CPU-bound. The performance challenge in this environment is to deal with a large volume of network I/O in an efficient manner.

At the same time, since I/O for these RDBMS applications comes from various sources (network, disk, local file-system and NFS), APIs for performance enhancements are also a critical factor.

Motivated by these goals and constraints, we have investigated a few kernel alternatives to UDP/IP, such as PF_PACKET. Our investigation has used micro-benchmarks such as netperf, and there is an ongoing effort into using PF_PACKET usage in Inter Process Communication (IPC) libraries for actual transaction-oriented RDBMS workloads.

In addition to the actual numbers themselves, interesting points that emerged during our investigation were the gaps between micro-benchmarks and real-world usage, critical features that directly impacted real workloads, and practical factors that impacted deployment for any solution.

The remainder of the paper is organized as follows. We first describe two types of environments commonly encountered in RDBMS deployments that are highly sensitive to network latency and the factors affecting performance in these environments. We provide an overview of the application constraints on APIs and tuning parameters offered by performance accelerating schemes in these environments. The Oracle clustering environment currently uses UDP[8] and RDS[9] sockets. We are currently evaluating the usage of PF_PACKET sockets in the Cluster. We describe the micro-benchmark and test-suites used for this evaluation and share the current results from our benchmarking effort as well as ongoing work in this space. Finally, we share some thoughts for ways to improve Linux Kernel stack latency that are currently under investigation.

Latency sensitive use-cases encountered in RDBMS

There are two types of use-cases in RDBMS environments that are sensitive to network latency.

1. Cluster applications offering services in a distributed computing environment. These services are CPU-bound, request-response transactions involving UDP flows that can be clearly identified by a 4-tuple.
2. Extract Transform Load (ETL) [2]. Here the input comes in as raw data in JSON or comma-separated values. The Compute Node converts the input to a Relational Database format that is stored to disk. The conversion to the Relational Database format is a CPU-intensive transform that preserves all the information while compressing the amount of data to be stored to the disk. The challenge in these environments is to find the right balance between CPU cycles needed for the transform itself, while also keeping up with the input rate coming in over the network

A notable aspect of these environments is that the performance critical flows involve packets sized close to, or larger than, the MTU of the link. For example, the Distributed Lock Manager is a typical Cluster service the performance-critical

traffic involves client requests of 512 bytes, with responses that are usually 8192 bytes. Although the complexity of improving network latency of small (64 byte) packet flows tends to be the focus of the common benchmarking and performance investigations, the challenges at the the opposite end of the spectrum, namely, improving performance for large packets. is less well-understood.

The ETL and cluster use-cases differ in the type of transport protocol and socket API used for networking. Cluster services tend to be stateless, with un-connected datagram sockets, whereas the ETL traffic comes over a stateful TCP connection. UDP based cluster services manage user-space state involving sequence number management and acknowledgement tracking with retransmissions to ensure guaranteed, reliable, ordered delivery over unconnected datagram sockets. The stateless nature of the cluster services, and the intrinsic simplicity of the UDP protocol, renders them more amenable to techniques that attempt to bypass the kernel protocol implementations. As a result of this observation, the focus of this benchmarking study was on UDP based cluster applications typified by the Lock Management Server.

Lock Management Server

The Lock Management Server (LMS) is a service provided by the Oracle Real Applications Cluster. The LMS is a Distributed Lock Manager that is implemented as a set of processes in the cluster which handle transaction oriented exchanges with clients that wish to obtain read-only locks on specific buffers owned by back-end database instances. Acquisition of the lock from the LMS is the bottleneck for the Database Transactions. Reducing the latency of LMS transactions is thus critical to system performance. The servers listen on a dynamically determined range of ports, and an incoming client request is assigned to a server based on a hash of fields in the UDP payload. The client is blocked until the server's response is received. In addition, the client has to process the server's response before it can send the next request. The interval between subsequent client requests is thus variable, and client input in this environment tends to have a bursty profile.

The server is the actual bottleneck for system performance in the LMS environment. Computations at the server are CPU bound and occur as follows. The server has the targeted buffer block loaded into its cache in the steady state. A single database instance holds the exclusive lock on the buffer in this state. The block can only be modified by the owner of the exclusive lock. The LMS will have to make a consistent read-only copy of the block when it is requested by a client. The read-only copy is sent back as the server response to the client's request.

All of the above computations are CPU intensive. In addition, since the clients assigned to a given server instance are blocked until the current request is serviced, the latency seen by the clients is gated by the server-side latency. Common techniques for improving network I/O performance include

- I/O batching
- reduction of system call overhead
- efficient management of context switches

The applicability of each of these techniques to the LMS environment is discussed below.

I/O batching Batching of network input allows the application to receive multiple client requests efficiently, and process them as a batch, instead of processing one request at a time, with a context switch per request.

For the LMS model, receive-side batching is easily adaptable to the application paradigm. When the LMS is woken from a `poll()`, `select()` or `epoll`, it begins reading packets from the file descriptor until it either runs out of input, or runs out of buffers into which it can process the input. A typical LMS server has about 64 clients hashing to its service port, so that the likelihood of finding multiple packets waiting at the input queue is high. Thus batching on the receive side is beneficial to system performance.

Batching of outgoing responses is a more complicated task. Since a client is blocked until the response to an outstanding request comes back, and the client cannot send the next request until the response is processed, inefficient transmit-side batching by the server can aggravate burstiness in the network traffic, resulting in sub-optimal system performance.

The LMS implementation that we used for our study implemented receive-side batching of input, but did not batch the outgoing responses,

Reduction of system-call overhead Current deployments of LMS use UDP and RDS transport for inter-process communication. Both of these transports involve one `sendmsg()` or `recvmsg()` call per I/O operation, thus triggering the associated system call overhead per packet.

Performance boosting methods such as NETMAP[11] and PF_PACKET[7] allow the application to use shared memory buffers with the kernel and eliminate the need for the system call per packet. In addition, the Linux kernel offers the `recvmmsg()` system call since Linux 2.3.33, that allows the application to read a batch of input datagrams in one system call. Both of these techniques were investigated as part of the benchmarking effort.

Efficient management of context switches The LMS server has to switch between CPU-intensive back-end computations for servicing the client's request, and CPU cycles to process network I/O. As a consequence of the receive-side batching model, when the server runs out of network input it falls back to `poll()`. This results in a context switch that allows the back-end computation to progress. The approach of having a dedicated CPU for network I/O and a different CPU for back-end computation has the drawback that it would result in cache-miss latencies and would not mitigate the performance problem. The ability to dynamically adjust the batch-size to handle the input receive rate can result in improved system performance. In our study, techniques such as PF_PACKET with TPACKET_V3 allowed us to explore this possibility.

Constraints imposed by the Clustering environment

The clustered services environment runs a number of services and applications that must co-exist. This requirement places

some constraints on any system tuning or feature modifications that may be used for improving network performance and latency.

Some of the services in this environment may be networking services that are not directly related to the database software, e.g., software for running SMTP. Thus global system tuning to improve the performance for one flow-pattern must not cause regressions in the performance of these other networking services.

Our database clustering services work through a shim library called `IPCLW` that is the intermediary between the database application and the I/O system calls. The `IPCLW` library allows the database applications to transparently switch between different forms of IPC, such as UDP, RDS [9] using both TCP and IB as the transport. The library provides this flexibility by assuming that the underlying system-call APIs have POSIX-compatible signatures. A ramification of these assumptions is that it places some constraints on performance accelerating kernel APIs that may be considered for inclusion into the `IPCLW` library. These constraints are listed below.

1. The database applications get I/O from multiple sources, such as disk, NFS and network. As a result, the software construct for receiving network I/O must be on a handle that is similar to a typical UNIX/POSIX file-descriptor or socket that can be added to a `select()`, `poll()` or `epoll()` set so that the application can multiplex between multiple sources of I/O without resorting to CPU intensive busy-poll loops
2. Software solutions for accelerating I/O must not require a major change in the application's threading model. The solution itself may need custom calls for handle creation, and may require additional wrappers around `sendmsg()` or `recvmsg()`. These can be accommodated in the database/cluster libraries via IPC-specific function callbacks. However radical changes to thread models that need a full-rewrite of the `IPCLW` library are not acceptable.
3. Accelerating the latency of a subset of flows must not occur at the expense of regressed latency for other packet flows. The solution must co-exist harmoniously with the existing Linux kernel stack implementations for flows corresponding to other network protocols that are not the target of the accelerated path.
4. Packet sharing with other Linux applications like tcpdump must be supported, since these applications may be needed for debugging Production issues.

Due to these constraints some of the popular solutions currently available for improving networking performance could not be considered as candidates for our study. The Data Plane Development Kit DPDK [5] provides a programming framework for Intel X86 processors that improves latency by using poll-mode user-space drivers. However the "Environment Abstraction Layer" (EAL) provided by DPDK does not provide a `select()`able socket, and the network handles for DPDK are not POSIX-compatible. An additional drawback is that DPDK radically modifies the threading model, making it unsuitable for incorporation into `IPCLW`.

One significant issue with DPDK is that the paradigm it offers is based on the premise that all packets coming in on the network interface will be processed by a single user-space application. Thus, the expectation is that a hypothetical user-space LMS using DPDK would consume the UDP flows relevant to the benchmark, but it would also need to make sure that other flows, e.g., NFS, SMTP or ARP, are also properly dispatched. DPDK provides the following methods to utilize the Linux kernel implementation paths for these protocols.

- Reinject these packets from user-space back into the kernel. DPDK supports this via the KNI paradigm [1] but this path is known to cause severe regressions in latency
- Use SRIOV to set up virtual functions (VF) with Ethernet filters to allow specific UDP flows to be passed up on a VF

Latency regressions from KNI are not acceptable in the cluster. The SRIOV solution requires complex configuration of the network interfaces of each system and manipulation of control-plane state such as ARP tables, reverse-path filters.

Similar considerations are associated with NETMAP [11], which provides a fast framework for packet I/O that bypasses the Linux kernel stack and provides a memory-map of the drivers packet buffers to user-space. The NETMAP API provides a `select()`able file-descriptor and meets some of the POSIX expectations of the `IPCLW` library. However, like DPDK, NETMAP passes up all packets on a NIC to the user-space NETMAP application, so that the Linux stack coexistence problem remains. Although there have been recent commits to allow packet sharing via host-rings, the solutions are not seamless.

In comparison, the PF_PACKET API in the Linux kernel provided many of the benefits of NETMAP via a clean socket API at the cost of some extra overhead of one extra `memcpy` from driver ring to `sk_buff`. Preliminary comparative benchmarking of request-response transactions using Netperf [6] did not show us any significant benefits to using NETMAP over PF_PACKET. Especially in consideration of the fact that PF_PACKET was already in the Linux kernel, we focused our benchmarking effort for this study on PF_PACKET.

PF_PACKET Overview

Linux kernels allow applications with administrative (i.e., CAP_NET_ADMIN) privileges to read and write fully formed Layer-2 frames using sockets of the PF_PACKET family. PF_PACKET sockets thus allow privileged applications to bypass the kernel protocol modules such as those for TCP, UDP and ICMP. The PF_PACKET socket provides a POSIX-compatible socket API to the application, meeting the constraints placed by the `IPCLW` environment. An additional advantage of the PF_PACKET socket is that it can be used in conjunction with shared memory-mapped buffers to avoid a user-kernel copy [7]. This shared memory-mapped ring buffer is made available through the TPACKET extensions to the PF_PACKET API.

An application that wishes to use the PF_PACKET API with TPACKET extensions has to set up a circular ring buffer of non-swappable memory in the kernel by invoking `setsockopt` to set the PACKET_TX_RING or PACKET_RX_RING socket option on the PF_PACKET

socket. The ring buffer that gets set up by this socket option is a series of blocks where each block is comprised of frames. The application then invokes `mmap()` to set up the shared memory mappings for this ring buffer. Subsequently, the application can read and write frames to the ring buffer.

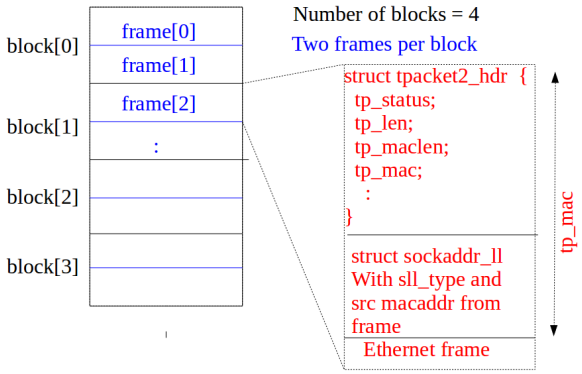

Figure 1: TPACKET_V2 extensions of the PF_PACKET socket

Figure shows the layout of the ring buffer for the PF_PACKET socket with the TPACKET_V2 format. Data exchanged between the application and the kernel is formatted with meta-data that is represented by the `tpacket2_hdr` for TPACKET_V2, and an analogous `tpacket3_hdr` for TPACKET_V3. The `tp_status` field in this header is used to indicate the current "owner" of the frame. After the kernel has populated the frame, the `tp_status` is set to TP_STATUS_USER, so that the application can safely read the frame. After it has completed the processing the frame, the application should set the `tp_status` to TP_STATUS_KERNEL to indicate that the kernel may now re-populate the frame with new data.

The shared memory ring-buffer is a lock-less way to do zero-copy reads and writes between user and kernel space. The optimization that it provides comes at the cost of a slight divergence from traditional POSIX APIs and assumes that the application can actively participate in the manipulation of the `tp_status` state.

The TPACKET_V3 extension to the PF_PACKET API provides additional optimizations over TPACKET_V2 by providing more application control over batching on the ingress packet path. The kernel triggers `->sk_data_ready()` to wake up a TPACKET_V2 in `poll()` as soon as a single frame is available for reading, but will wake up a TPACKET_V3 application only if one of the following two conditions are satisfied:

- an entire block of frames is available for reading, or,
- a block is partially filled, but an application defined timeout has expired

The application can specify the number of frames in a block, as well as the desired timeout, as part of the socket options to set up the transmit and receive rings. As we show in our results below, controlling these parameters to match the anticipated packet input rate helped our application achieve optimal system throughput with good CPU utilization.

Benchmarks

Our study focused on the evaluation of IPC transports used in Database Clustering environments using the benchmarks described below.

1. Typical networking micro-benchmarks like Netperf [6],
2. micro-benchmarks associated with Cluster IPC libraries (`IPCLW`) used by the Oracle Database software,
3. cluster benchmark suites that simulate database workloads and instrument the `IPCLW` library performance.

A detailed description of each of these benchmarks follows.

Micro-benchmarking with Netperf We evaluated three types of IPC transport mechanisms using Netperf request-response micro-benchmarks. These were

- UDP sockets using `sendmsg()` and `recvmsg()` [12],
- UDP sockets using the `recvmmsg()`,
- PF_PACKET sockets with TPACKET_V2 and TPACKET_V3

The `recvmmsg()` system call [10] is an extension of the POSIX `recvmmsg()` call that allows the caller to batch the reception of multiple messages on a socket at the expense of a single system call.

Our micro-benchmarking used the Netperf UDP_RR (UDP request-response) tests with a stock implementation of the `netperf` client. We replaced the standard `netserver` with our own server implementation that could switch between the three transports listed above. The server supported `sendmsg()` and `recvmsg()` for in a simple batching loop by implementing the pseudo-code below.

```
err = poll(..); /* wait for input */
while (/* recvfrom() returns a req */) {
      sendto( /* response */
}
```

The `recvmmsg()` version for also implemented a similar loop with the difference that the `recvfrom()` call was replaced by a call to `recvmmsg()` using a batch-size of 64 packets and a timeout of 10 ms. Explicit system calls for sending and receiving packets are not required in the PF_PACKET/TPACKET paradigm. Receive-side batching with PF_PACKET was available as a ramification of the shared memory ring buffer between user and kernel. Thus all three transports were conceptually equivalent in functionality.

Our test used 64 `netperf` clients per-server, based on the approximate client-to-server ration observed in our Cluster environments in Production. The size of the request packet used was 512 bytes, with response packet size set to 1024 bytes. The test server in our environment was an Intel X86 system with an Intel E7 CPU and 256G RAM. The test network interface driver was an Intel Ethernet Controller XL710 running at 40 Gbps. Results reported in Figure 2 were obtained after configuring the UDP Receive Side Scaling (RSS)

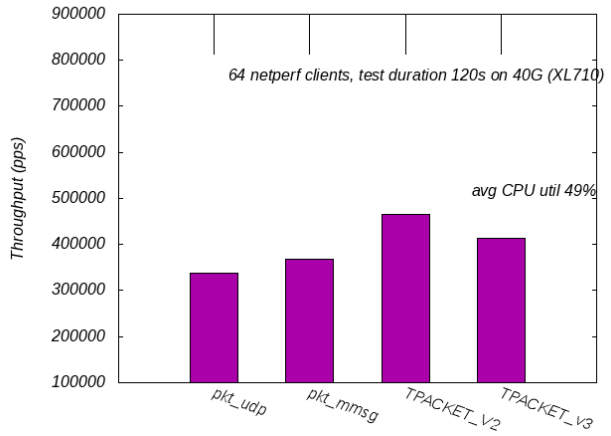

Figure 2: Netperf throughput results

to spread traffic across CPUs with UDP port numbers included for RSS hashing based on the description for 10Gbps adapters provided in [4]. No additional tuning was applied.

Figure 2 reports the throughput in packets per-second as instrumented by the driver's hardware counters for the four types of transports that we investigated. The TPACKET_V2 method reported the best throughput numbers at 370K pps and provided a significant improvement over UDP sockets. In the case of UDP and TPACKET_V2 transports, CPU utilization was at 100% when the maximum throughput reported in Figure 2 was achieved. We found that the maximum possible throughput with TPACKET_V3 could match or better the value for TPACKET_V2 with CPU utilization at 100%, however the graph shows the optimal system performance point where the CPU utilization was at 49% with improved throughput in comparison with UDP sockets. We arrived at this optimal performance point after some experimentation with the tuning of batching parameters.

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

Table 1: TPACKET_V3 batching behavior.

As previously described on Page 4, the TPACKET_V3 version of the PF_PACKET API allowed the application to customize batching parameters such as number of frames per block. Table 1 records the trends that we observed when the number of frames per block (`fpb`) was modified for a fixed value of the timeout (10 ms) in an environment involving 64 clients per single server thread. All 64 clients were active at steady state. With 16 frames per block, the server would always find a ready block and frame for processing. Thus the server was able to keep the CPU 100% busy and process the maximum throughput at `fpb = 16`.

The other end of the spectrum had 64 frames per block, with input provided by 64 clients. Even at steady state, there was a finite latency gap for filling a block after which the server could be woken up from `poll()` to process the block. The serialized processing in this model resulted in lower throughput, and degraded CPU utilization.

A good balance was achieved at `fpb=32` where the system was able to keep the server busy processing a block of 32 frames while the kernel was filling up the next block to be passed up to the server. The optimal system performance was achieved when the value of `fpb` is approximately half the value of client-to-server ratio. Thus, when the number of clients was increased to 128, optimal system performance was achieved at `fpb=32`.

A conclusion that can be drawn from this observation is that the optimal values of system performance can be achieved by dynamically adjusting the batching parameters based on a statistical sampling of the client input rate. Extending the PF_PACKET API to accommodated such dynamic tuning may be a valuable feature to add to the API.

Cluster-specific Benchmark Suites As described earlier, clustering applications in the Oracle "Real Applications Cluster" use a library (`IPCLW`) that acts as the shim between the database application and the IPC specific system calls for I/O that are native to the Operating System. The `IPCLW` library allows the application to switch between various types of transport such as RDS-TCP [9] and UDP. At the time of this writing, we are also working on adding support for PF_PACKET sockets to the `IPCLW` library.

The `IPCLW` library is instrumented using a database specific test suite, CRTEST, that simulates a real database workload for various types of clustering services like LMS. The CRTEST test suite is a set of Cluster atomic benchmark tests that simulates a Lock Management Server described earlier.

The CRTEST suite can be run with a varying number of clients, and is typically run with $1, 2, 4, \ldots, 64$ clients. For each value of the number of clients ($nclients$) the test instruments the throughput and latency for network I/O

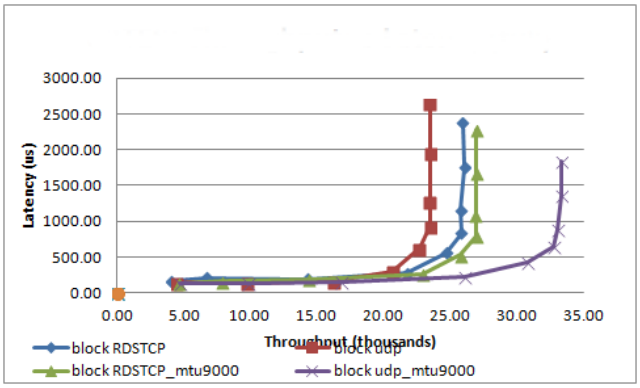

Figure 3: CRTEST Throughput vs Latency results

The primary objective when running the CRTEST suite was to evaluate the performance of RDS-TCP in comparison to UDP. A secondary objective was to attempt to use Jumbo frames as a substitute for UDP Fragmentation Offload. Figure 3 shows the throughput and latency at each value of $nclients$.

The throughput shown in this figure is the value in thousands of 8K blocks per second. Results were obtained on an Intel X86 system with an E5 CPU. The NIC used for the test was the Intel 82599ES 10Gbps driver.

The results in Figure 3 highlight some characteristic properties of this particular client-server environment.

The shape of the graph for each transport has a pattern of an almost horizontal segment which abruptly becomes a vertical line (the "wall") at which throughput reaches the upper-bound for that transport. This shape can be explained as follows.

As the the number of clients per single-threaded server was increased, the throughput initially increased for the same latency, contributing to the horizontal segment. This trend continued until the limit of the server's processing capacity was reached. At that point, adding additional clients caused queue build-up and thus increased latency, for the same throughput. This results in the "wall" for that transport type and MTU in Figure 3.

The limiting bound on throughput for RDS-TCP was higher than the limit for UDP indicating that RDS-TCP had better server-side capacity. Both RDS-TCP and UDP-based transports in our model were required to provide a reliable datagram delivery model to the application. The kernel TCP state engine in the kernel ensured guaranteed, reliable, ordered delivery for RDS-TCP. In the case of the UDP-based transport, the application managed the needed state to achieve reliable delivery. This state was comprised of sequence numbers, acknowledgements and timeout-based retransmits for each datagram. The UDP based model tracked this client-sever state at each client whereas the single kernel TCP state machine tracked the equivalent state for an IP address pair with RDS-TCP. The UDP model also intrinsically had a higher vulnerability to scheduling delays in user-space at the server. These delays caused ACKs to be delayed, spurious retransmits and other issues that resulted in the lower bound on the maximum throughput possible with UDP.

Both throughput and latency improved significantly for UDP when Jumbo frames were used. Latency for 64 clients went from 2600 μs to 1800 μs, and throughput increased from 22K to 35K blocks per second. RDS-TCP also showed some improvement, though the gain was less than with UDP. UDP with Jumbo packet size had a much larger gain due to the impact of IP fragmentation and reassembly on the UDP/IP stack. The UDP protocol layer is mostly stateless in comparison with TCP, and the bulk of the work in sending/receiving an 8K block is done at the IP fragmentation and reassembly layers of the UDP/IP stack. By enabling Jumbo frames we removed this work from the packet-path and were able to release additional CPU cycles for use by the LMS server. Reduction in latency and CPU utilization in the network stack thus resulted in the improved latency and throughput noted in the graph in Figure 3.

In comparison TCP, with TCP Segmentation Offload (TSO), was able to send large frames to the driver even with 1500 byte MTU. In addition TCP had to manage a significant amount of protocol state even in the absence of congestion. That state remained the same, regardless of interface MTU. Thus the benefits of going to Jumbo MTU were smaller than with UDP.

Fragmentation of an 8K sized UDP datagram to MTU sized frames is unavoidable work that has to be handled somewhere in the packet output path. This work could be done at the application level, in the Linux kernel, or via UDP fragmentation offload. In order to retain the benefits of Equal Cost Multipath (ECMP) routing, it is desirable to break up the 8K bytes of data into smaller MTU-sized UDP packets so that each packet has a copy of the UDP header that can be used by intermediate routers for flow multiplexing. However, this would require the application to manage its own message boundaries which then logically leads to an architecture that resembles RDS-TCP. Leveraging the kernel implementation of IP fragmentation requires CPU cycles and it would be preferable to be able to offload this computation to the NIC, similar to TCP Segmentation Offload. The challenges for UDP Fragmentation Offload (UFO) are the difficulty in computing the UDP checksum of very large (e.g., 65K) sized packets, and tracking state at the receiver for IP reassembly. For these reasons, UFO is not commonly supported on NICs.

As a result of these considerations, our current approach to mitigating the overhead of IP fragmentation and reassembly is to enable Jumbo across the Cluster and Datacenter. This solution is sufficient for controlled environments, though it does not scale for packets that traverse long internet paths.

Gaps In Micro-benchmarking vs Production Testing

Our investigation helped us recognize many aspects of performance analysis which highlighted the gaps between results obtained with micro-benchmarks in comparison to those under real-world workloads. We summarize some of these findings below.

- **Constraints on system tuning.** Common tuning recommendations such as disabling iommu, disabling Ethernet flow-control, or setting sysctl tunables that improve performance for connected UDP sockets could not be used in our environment where the servers in our test were running many different features, some of which would have been impacted negatively by such tweaks
- **Cost of bypassing kernel IP modules** We focused our investigation on connectionless UDP sockets based on the observation that it was a stateless packet-exchange protocol, and thus easy to implement in user-space via PF_PACKET sockets. We found that even this light-weight packet exchange required some additional overhead that had to be carefully managed in a user-space implementation. The `IPCLW` library had to correctly set up the Ethernet destination address by consulting the current IP forwarding table (FIB) and ARP tables for optimal switching of the Ethernet in all network topologies. The `IPCLW` library cached this information for each destination IP address as a performance optimization, but then had to manage a NETLINK listener to stay synchronized with kernel control-plane state. In addition, as discussed in the previous section, IP fragmentation and reassembly were an unavoidable cost in the packet transmit path.
- **Zero-copy and shared memory buffers change the user API.** An important consideration in selecting PF_PACKET

sockets for our study was the availability of the shared memory ring buffer feature between user-space and kernel for packet I/O. The shared memory avoided one `memcpy` user-kernel boundary and had the potential of supplying a significant performance boost when used in conjunction with batched send/receive. This observation had also been noted by the authors of NETMAP [11].

In practice we found that incorporating zero-copy APIs into application libraries was not seamless. The `IPCLW` library follows indirections with fixed signatures for input and output over all transports. In the case of non-RDMA transports, this signature is expected to conform to the POSIX definitions of `sendmsg()` and `recvmsg`. We were able to add support in the `IPCLW` library to provide customized, ring-aware versions for these indirections to handle PF_PACKET sockets without too much effort. However, since the calling application is unaware of the underlying shared memory semantics for this IPC method, it also does not have the context of "ownership" via TP_STATUS_KERNEL/TP_STATUS_USER [7]. Thus the shim function at the time of this writing ends up doing an explicit `memcpy` from the ring buffer to the application's buffer, detracting from the performance benefits of PF_PACKET. A possible way to avoid this `memcpy` overhead is by changing the PF_PACKET API in `IPCLW` to resemble the RDMA paths supported in the library, and this is currently under investigation

Ongoing and Future Work

The PF_PACKET shared-memory ring architecture allows for a zero-copy interface at the user-kernel boundary. This optimization comes with the caveat that applications that wish to exploit these semantics now need to track buffer ownership state between user and kernel. We are currently evaluating the feasibility of evolving the PF_PACKET shared-memory paths in the `IPCLW library` to follow analogous APIs used for RDMA memory registration in Infiniband.

The memory copy from the kernel's `sk_buff` to the driver DMA ring remains after this optimization. Methods to eliminate that memory copy are under discussion at the time of this writing [3]. The approach being proposed to eliminate this memory copy is through the introduction of a device indirection via `ndo_ops` device drivers such that the packets received by HW can directly be transferred to/from the PF_PACKET ring via DMA.

Acknowledgments

Thanks to Yasuo Hirao for providing us access to the CRTEST benchmark results, Rick Jones for helpful hints during netperf navigation and Shannon Nelson for help with reviewing this document.

References

[1] DPDK Kernel NIC Interface. `http://dpdk.org/doc/guides/prog_guide/kernel_nic_interface.html`.

[2] Oracle Database Data Warehousing Guide . `https://docs.oracle.com/cd/B19306_01/server.102/b14223/ettover.htm`.

[3] RFC af_packet: direct DMA for packet interface. `http://patchwork.ozlabs.org/patch/720937/`.

[4] Linux* Base Driver for the Intel(R) Ethernet 10 Gigabit PCI Express Family of Adapters. `https://downloadmirror.intel.com/22919/eng/README.txt`.

[5] Intel Corporation. Intel Corporation. 2013a. Intel Data Plane Development Kit (Intel DPDK) Programmer's Guide. Reference Number: 326003-003.

[6] 2004. The Public Netperf Homepage. `http://www.netperf.org/netperf/NetperfPage.html`.

[7] PACKET socket interface - Linux kernel documenation. Documentation/networking/packet_mmap.txt in the Linux Kernel Source.

[8] Postel, J. 1980. User Datagram Protocol. RFC 768 (Internet Standard).

[9] RDS Reliable Datagram Sockets. `https://oss.oracle.com/projects/rds/dist/documentation/rds-3.1-spec.html`.

[10] recvmsg(2) - Linux man page . `https://linux.die.net/man/2/recvmsg`.

[11] Rizzo, L. 2012. Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, 9–9. Berkeley, CA, USA: USENIX Association.

[12] Stevens, W. R.; Fenner, B.; and Rudoff, A. M. 2003. *UNIX Network Programming, Vol. 1*. Pearson Education, 3 edition.

Author Biography

Sowmini Varadhan is a Consulting Software Engineer in the Virtual Operating Systems Group at Oracle Corporation. where she works on projects spanning Kernel Networking, Distributed Computing, and Performance. Tushar Dave is a Principal Linux Engineer in the Mainline Linux Kernel group at Oracle Corporation where he works on networking drivers, DMA, IOMMU and PCIe.