The Performance Analysis of Linux Networking – Packet Receiving

Wenji Wu, Matt Crawford

Fermilab

CHEP 2006

wenji@fnal.gov, crawdad@fnal.gov

Topics

- Background
- Problems
- Linux Packet Receiving Process
  - NIC & Device Driver Processing
  - Linux Kernel Stack Processing
    - IP
    - TCP
    - UDP
  - Data Receiving Process
- Performance Analysis
- Experiments & Results

1. Background

- Computing model in HEP
  - Globally distributed, grid-based
- Challenges in HEP
  - To transfer physics data sets – now in the multi-petabyte (10^{15} bytes) range and expected to grow to exabytes within a decade – reliably and efficiently among facilities and computation centers scattered around the world.
- Technology Trends
  - Raw transmission speeds in networks are increasing rapidly, the rate of advancement of microprocessor technology has slowed.
  - Network protocol-processing overheads have risen sharply in comparison with the time spend in packet transmission in the networks.

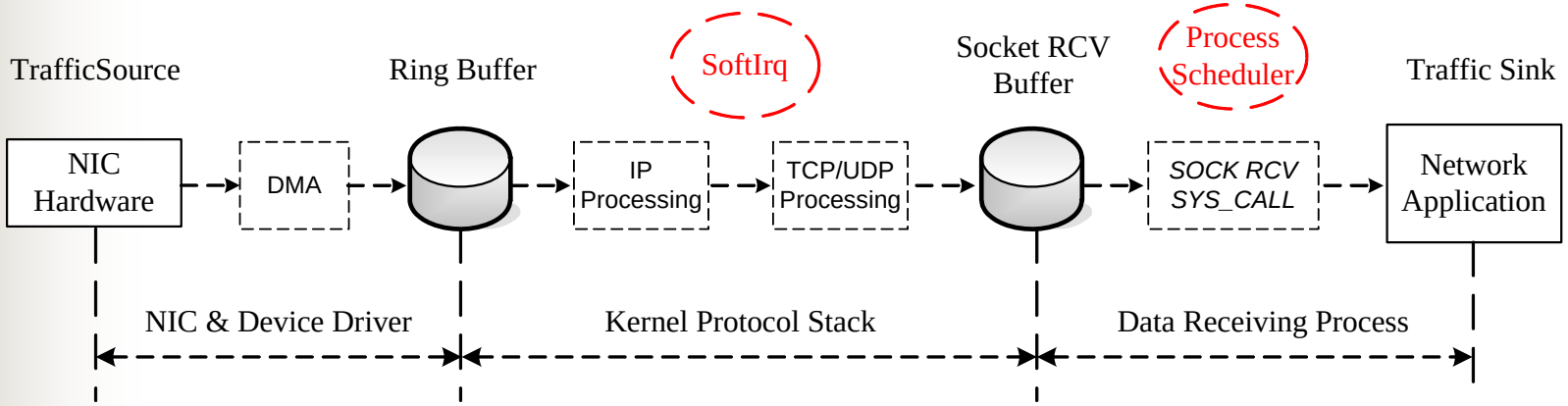
2. Problems

- What, Where, and How are the bottlenecks of Network Applications?
  - Networks?
  - Network End Systems?

We focus on the Linux 2.6 kernel.

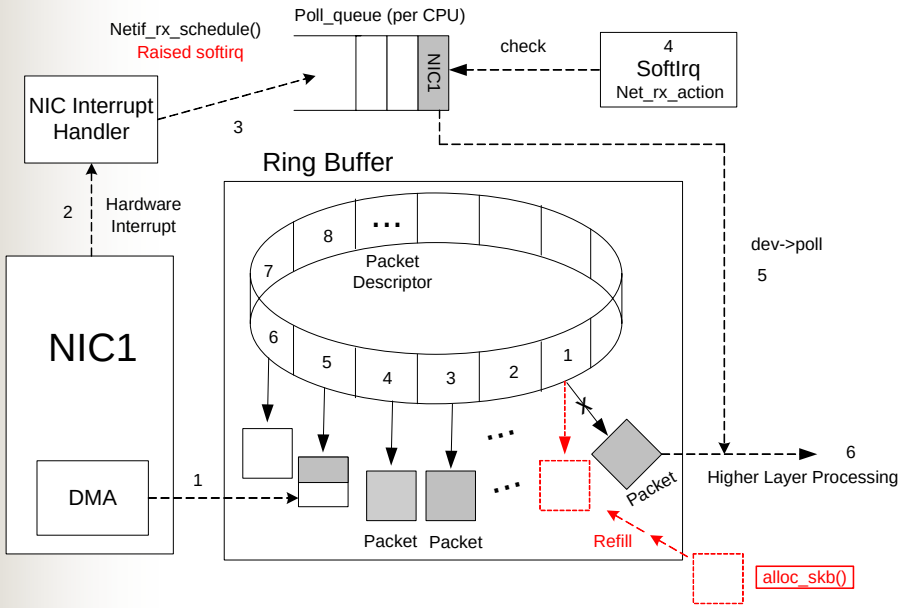
3. Linux Packet Receiving Process

Linux Networking subsystem: Packet Receiving Process



- Stage 1: NIC & Device Driver
  - Packet is transferred from network interface card to ring buffer
- Stage 2: Kernel Protocol Stack
  - Packet is transferred from ring buffer to a socket receive buffer
- Stage 3: Data Receiving Process
  - Packet is copied from the socket receive buffer to the application

NIC & Device Driver Processing



NIC & Device Driver Processing Steps

1. Packet is transferred from NIC to Ring Buffer through DMA
2. NIC raises hardware interrupt
3. Hardware interrupt handler schedules packet receiving software interrupt (Softirq)
4. Softirq checks its corresponding CPU's NIC device poll-queue
5. Softirq polls the corresponding NIC's ring buffer
6. Packets are removed from its receiving ring buffer for higher layer processing; the corresponding slot in the ring buffer is reinitialized and refilled.

- Layer 1 & 2 functions of the OSI 7-layer network Model
- Receive ring buffer consists of packet descriptors
  - When there are no packet descriptors in ready state, incoming packets will be discarded!

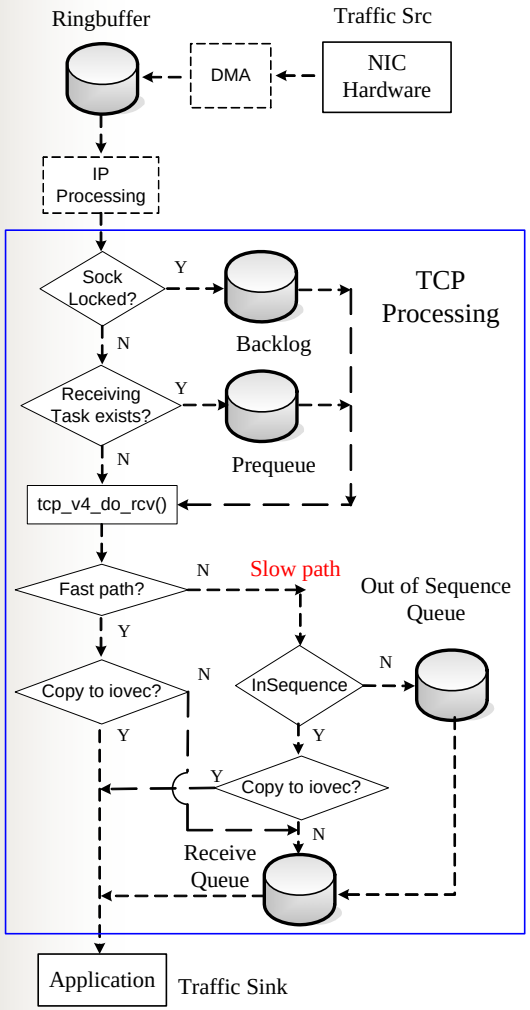
Kernel Protocol Stack – IP

- IP processing
  - IP packet integrity verification
  - Routing
  - Fragment reassembly
  - Preparing packets for higher layer processing.

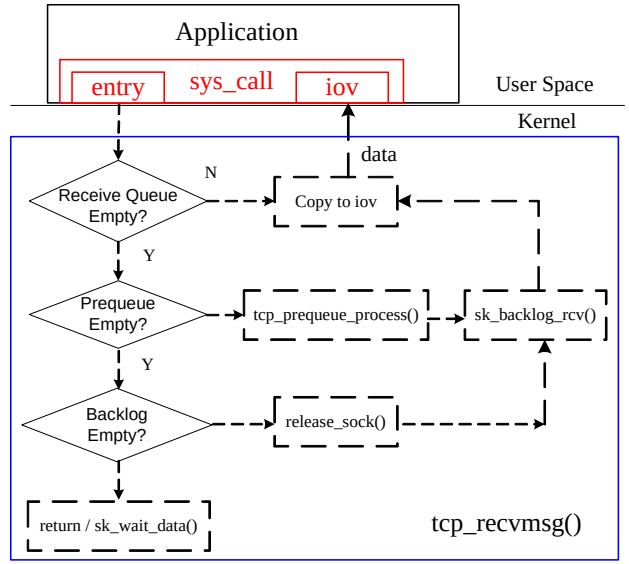
Kernel Protocol Stack – TCP 1

- TCP processing
  - TCP Processing Contexts
    - Interrupt Context: Initiated by Softirq
    - Process Context: initiated by data receiving process;
      - more efficient, less context switch
  - TCP Functions
    - Flow Control, Congestion Control, Acknowledgement, and Retransmission
- TCP Queues
  - Prequeue
    - Trying to process packets in process context, instead of the interrupt contest.
  - Backlog Queue
    - Used when socket is locked.
  - Receive Queue
    - In order, acked, no holes, ready for delivery
  - Out-of-sequence Queue

Kernel Protocol Stack – TCP 2



TCP Processing- Interrupt context



TCP Processing- Process context

Except in the case of prequeue overflow, Prequeue and Backlog queues are processed within the process context!

Kernel Protocol Stack – UDP

- UDP Processing
  - Much simpler than TCP
  - UDP packet integrity verification
  - Queue incoming packets within Socket receive buffer; when the buffer is full, incoming packets are discarded quietly.

Data Receiving Process

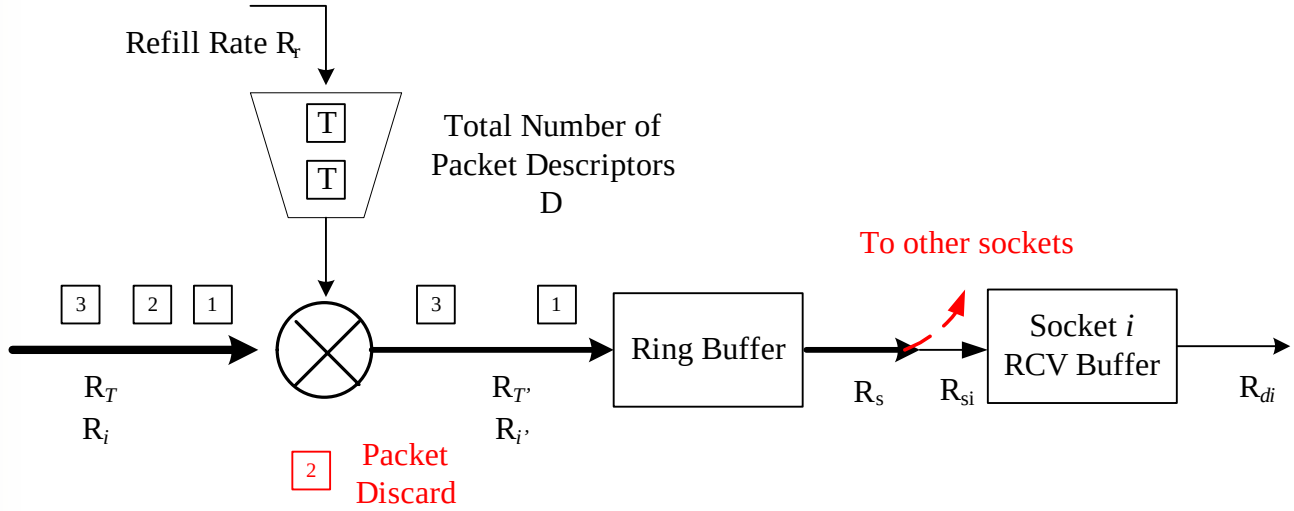
- Copying packet data from the socket's receive buffer to user space through *struct iovec*.
- Socket-related systems calls
- For TCP stream, data receiving process might also initiate the TCP processing in the process context.

4. Performance Analysis

Notation

$R_T(t)$, $R_{T'}(t)$:	Offered and accepted total packet rate (Packets/Time Unit);
$R_i(t)$, $R_{i'}(t)$:	Offered and accepted packet rate for data stream i (Packets/Time Unit);
$R_r(t)$:	Refill rate for used packet descriptor at time t (Packets/Time Unit);
D:	The total number of packet descriptors in the receiving ring buffer;
$A(t)$:	The number of packet descriptors in the ready state at time t;
$R_s(t)$:	Kernel protocol packet service rate (Packets/Time Unit);
$R_{si}(t)$:	Softirq packet service rate for stream i (Packets/Time Unit);
$R_{di}(t)$:	Data receiving process packet service rate for stream i (Packets/Time Unit);
$B_i(t)$:	Socket i's receive buffer size at time t (Bytes);
QB_i:	Socket i's receive buffer quota (Bytes);
λ:	Data receiving process' packet service rate when the process is running
$\tau_{\min}$:	The minimum time interval between a packet's ingress into the system and its first being serviced by a softirq;

Mathematical Model



- Token bucket algorithm models NIC & Device Driver receiving process
- Queuing process models the receiving process' stage 2 & 3

Token Bucket Algorithm – Stage 1

The reception ring buffer is represented as the token bucket with a depth of D tokens. Each packet descriptor in the ready state is a token, granting the ability to accept one incoming packet. The tokens are regenerated only when used packet descriptors are reinitialized and refilled. If there is no token in the bucket, incoming packets will be discarded.

$$\forall t > 0, \qquad R_{T'}(t) = \begin{cases} R_T(t), & A(t) > 0 \\ 0, & A(t) = 0 \end{cases} \tag{1}$$

To admit packets into system without discarding, it should have:

$$\forall t > 0 \ , \ A(t) > 0 \tag{2}$$

$$A(t) = D - \int_0^t R_{T'}(\tau) d\tau + \int_0^t R_r(\tau) d\tau \ , \ \forall t > 0 \tag{3}$$

NIC & Device Driver might be a potential bottleneck!

Token Bucket Algorithm – Stage 1

To reduce the risk of being the bottleneck, what measures could be taken?

- Raise the protocol packet service rate
- Increase system memory size
- Raise NIC's ring buffer size *D*
  - *D* is a design parameter for the NIC and driver.
  - For an NAPI driver, D should meet the following condition to avoid unnecessary packet drops:

$$D \geq \tau_{\min} * R_{\max} \qquad (4)$$

Queuing process – Stage 2 & 3

For stream *i*; it has

$$R_{i'}(t) \le R_i(t) \quad \text{and} \quad R_{si}(t) \le R_s(t) \qquad (5)$$

It can be derived that:

$$B_i(t) = \int_0^t R_{si}(\tau)d\tau - \int_0^t R_{di}(\tau)d\tau \qquad (6)$$

For UDP, when receive buffer is full, incoming UDP packets are dropped;
For TCP, when receive buffer is approaching full, flow control would throttle sender' data rate;

For network applications, it is desirable to raise (7)

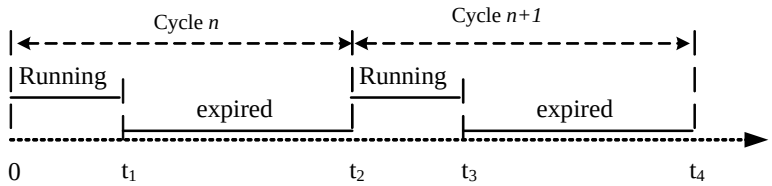
$$QB_i - \int_0^t R_{si}(\tau)d\tau + \int_0^t R_{di}(\tau)d\tau \qquad (7)$$

A full receive buffer is another potential bottleneck!

Queuing process – Stage 2 & 3

- What measures can be taken?
  - Raising socket's receive buffer size QB_i
    - Configurable, subject to system memory limits
  - Raising $R_{di}(t)$
    - Subject to system load and the data receiving process' *nice* value
    - Raise data receiving process' CPU share
      - Increase nice value
      - Reduce system load

$$R_{di}(t) = \begin{cases} \lambda, & 0 < t < t_1 \\ 0, & t_1 < t < t_2 \end{cases}$$

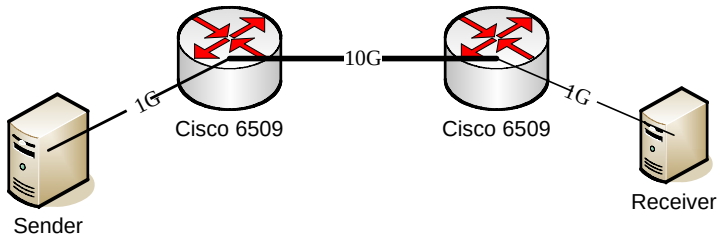


(8)

5. Experiments & Results

Experiment Settings

- Run *iperf* to send data in one direction between two computer systems;
- We have added instrumentation within Linux packet receiving path
- Compiling Linux kernel as background system load by running **make –*nj***
- Receive buffer size is set as 20M bytes



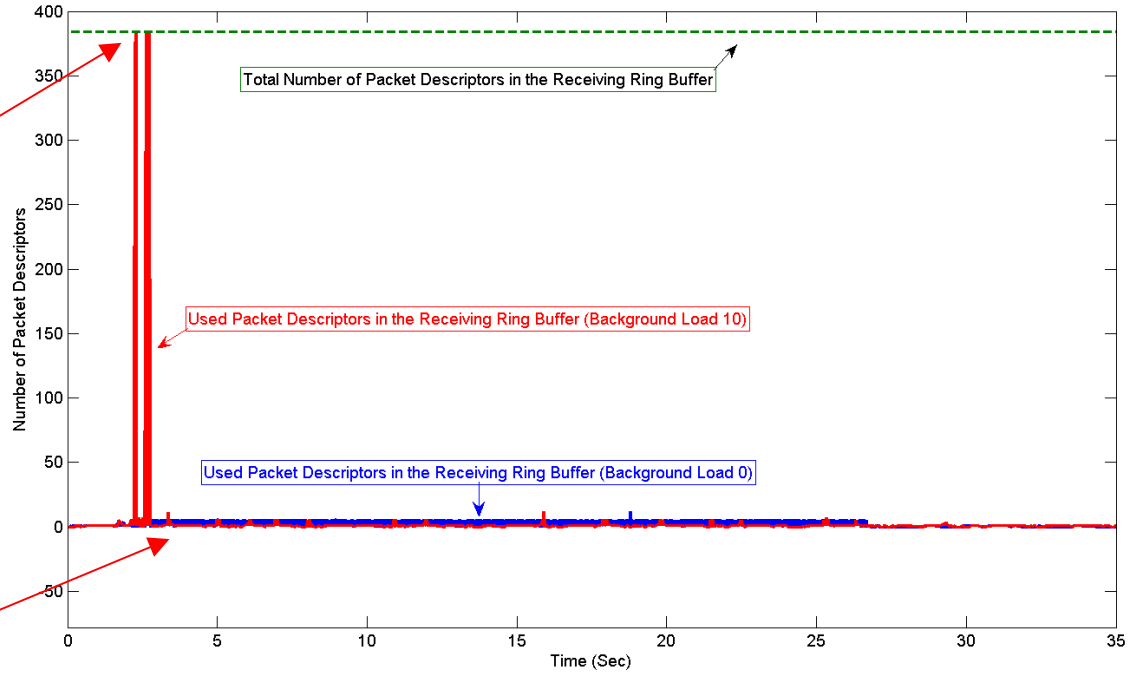
Fermi Test Network

	Sender	Receiver
CPU	Two Intel Xeon CPUs (3.0 GHz)	One Intel Pentium II CPU (350 MHz)
System Memory	3829 MB	256MB
NIC	Tigon, 64bit-PCI bus slot at 66MHz, 1Gbps/sec, twisted pair	Syskonnect, 32bit-PCI bus slot at 33MHz, 1Gbps/sec, twisted pair

Sender & Receiver Features

Experiment 1: receive ring buffer

Running out packet descriptors



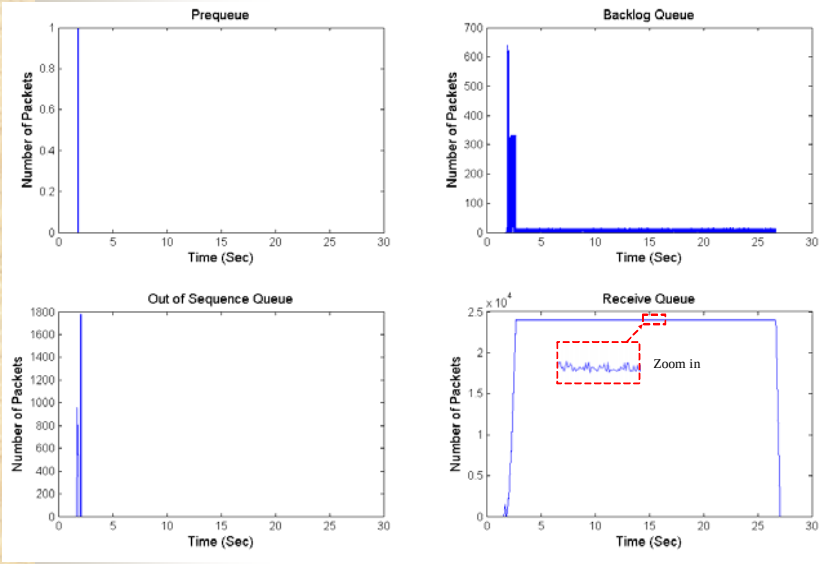
TCP throttles rate to avoid loss

Figure 8

Total number of packet descriptors in the reception ring buffer of the NIC is 384

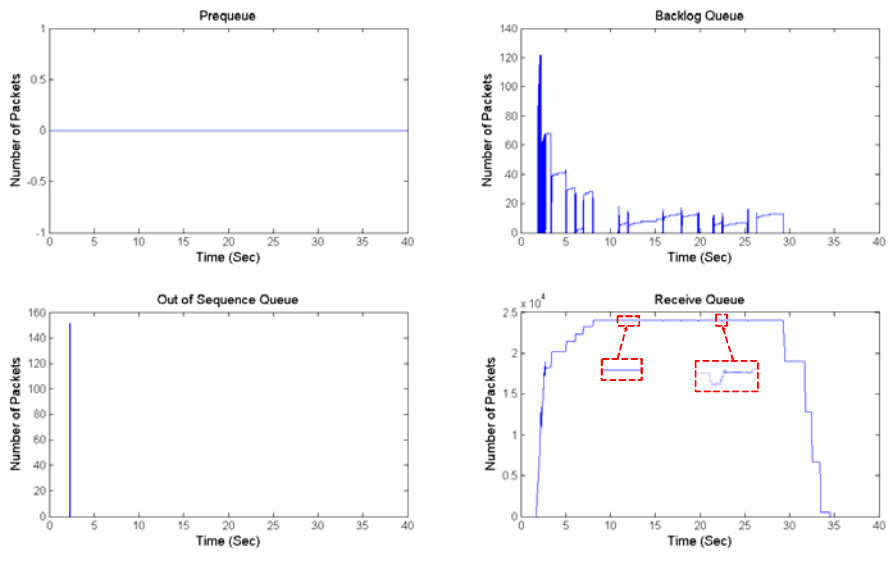
Receive ring buffer could run out of its packet descriptors: Performance Bottleneck!

Experiment 2: Various TCP Receive Buffer Queues



Background Load 0

Figure 9



Background Load 10

Figure 10

Experiment 3: UDP Receive Buffer Queues

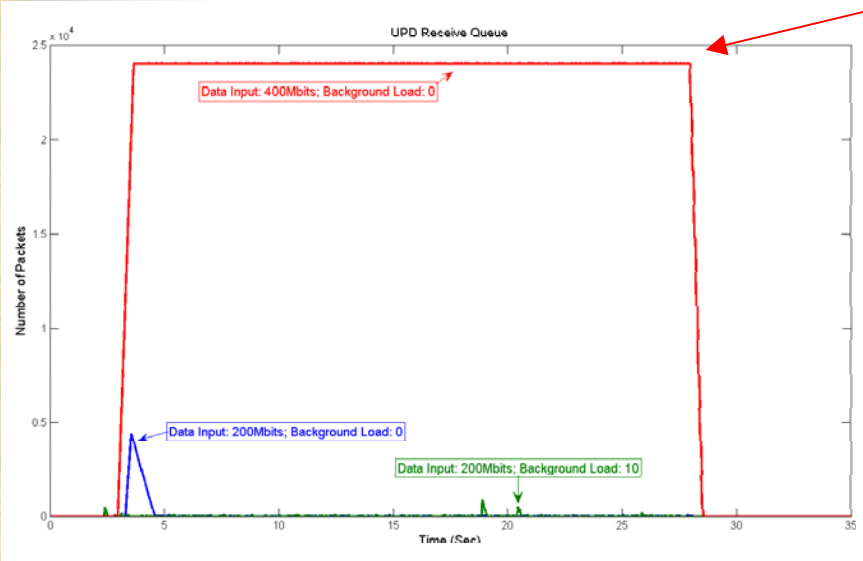
The experiments are run with three different cases:

(1) Sending rate: 200Mb/s, Receiver's background load: 0;
(2) Sending rate: 200Mb/s, Receiver's background load: 10;
(3) Sending rate: 400Mb/s, Receiver's background load: 0.

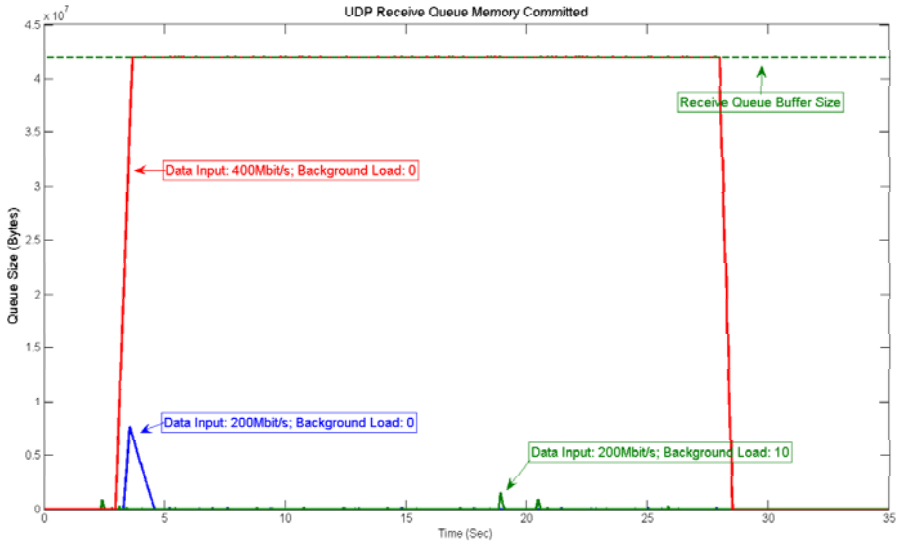
Transmission duration: 25 seconds; Receive buffer size: 20 Mbytes

Receive livelock problem!

When UDP receive buffer is full, incoming packet is dropped at the socket level!



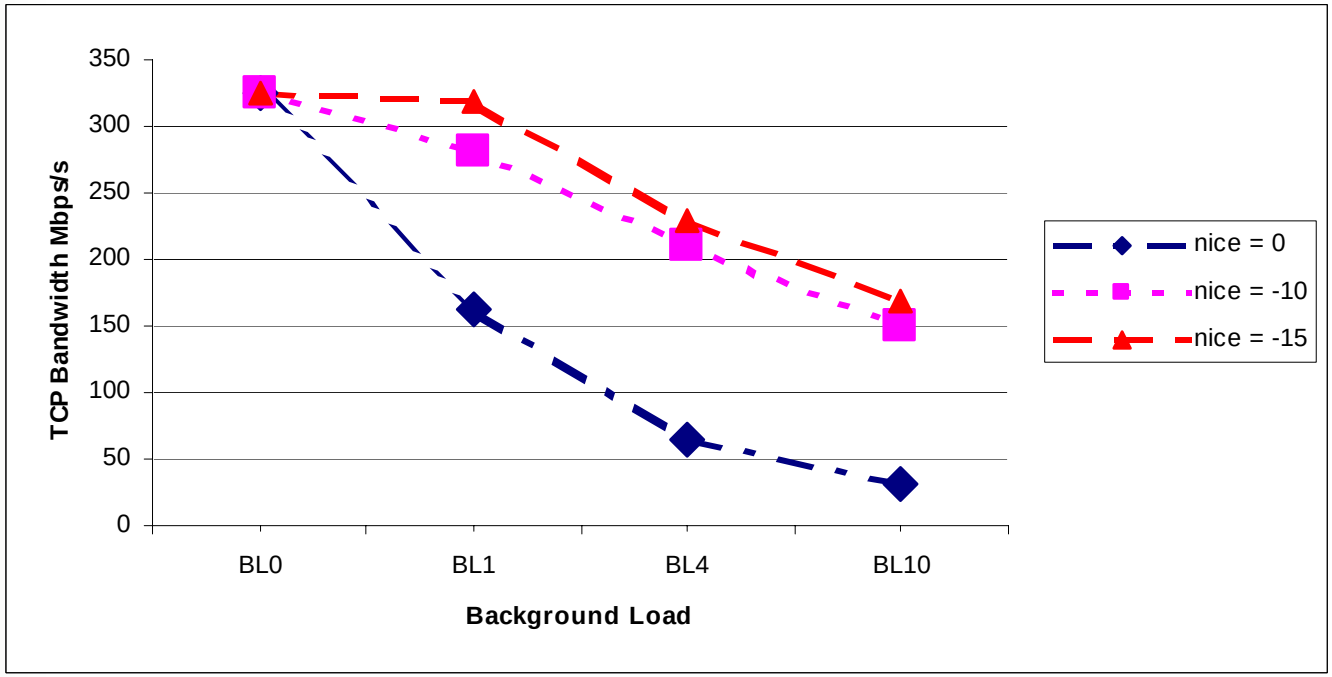
UPD Receive Buffer Queues
Figure 11



UDP receive Buffer
Committed Memory
Figure 10

Both cases (1) and (2) are within receiver's handling limit. The receive buffer is generally empty
The effective data rate in case (3) is 88.1Mbits, with a packet drop rate of 670612/862066 (78%)

Experiment 3: Data receive process



Sender transmits one TCP stream to receiver with the transmission duration of 25 seconds. In the receiver, both data receiving process' nice value and the background load are varied. The nice values used in the experiments are: 0, -10, and -15.

Conclusion:

- The reception ring buffer in NIC and device driver can be the bottleneck for packet receiving.
- The data receiving process' CPU share is another limiting factor for packet receiving.

References

[1] Miguel Rio, Mathieu Goutelle, Tom Kelly, Richard Hughes-Jones, Jean-Philippe Martin-Flatin, and Yee-Ting Li, "A Map of the Networking Code in Linux Kernel 2.4.20", March 2004.

[2] J. C. Mogul and K. K. Ramakrishnan, “Eliminating receive livelock in an interrupt-driven kernel”, ACM Transactions on Computer Systems, 15(3): 217--252, 1997.

[3] Klaus Wehrle, Frank Pahlke, Hartmut Ritter, Daniel Muller, and Marc Bechler, The Linux Networking Archetecture – Design and Implementation of Network Protocols in the Linux Kernel, Prentice-Hall, ISBN 0-13-177720-3, 2005.

[4] www.kernel.org

[5] Robert Love, Linux Kernel Development, Second Edition, Novell Press, ISBN: 0672327201, 2005.

[6] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, Linux Device Drivers, 3rd Edition, O’Reilly Press, ISBN: 0-596-00590-3, 2005.

[7] Andrew S. Tanenbaum, Computer Networks, 3rd Edition, Prentice-Hall, ISBN: 0133499456, 1996.

[8] Arnold O. Allen, Probability, Statistics, and Queueing Theory with Computer Science Applications, 2nd Edition, Academic Press, ISBN: 0-12-051051-0, 1990.

[9] Hoskote, Y., et.al., A TCP offload accelerator for 10 Gb/s Ethernet in 90-nm CMOS, Solid-State Circuits, IEEE Journal of Volume 38, Issue 11, Nov. 2003 Page(s):1866 – 1875.

[10] Regnier, G., et.al., TCP onloading for data center servers, Computer, Volume 37, Issue 11, Nov. 2004 Page(s):48 - 58

[11] Transmission Control Protocol, RFC 793, 1981

[12] http://dast.nlanr.net/Projects/Iperf/