To Appear in The International Journal of Computer Communications, Elsevier, 2006
DOI information: 10.1016/j.comcom.2006.11.001

The Performance Analysis of Linux Networking – Packet Receiving[*]

Wenji Wu[+], Matt Crawford, Mark Bowden
Fermilab, MS-368, P.O. Box 500, Batavia, IL, 60510
Email: wenji@fnal.gov, crawdad@fnal.gov, bowden@fnal.gov
Phone: +1 630-840-4541, Fax: +1 630-840-8208

Abstract

The computing models for High-Energy Physics experiments are becoming ever more globally distributed and grid-based, both for technical reasons (e.g., to place computational and data resources near each other and the demand) and for strategic reasons (e.g., to leverage equipment investments). To support such computing models, the network and end systems, computing and storage, face unprecedented challenges. One of the biggest challenges is to transfer scientific 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. Both the network and end systems should be able to provide the capabilities to support high bandwidth, sustained, end-to-end data transmission. Recent trends in technology are showing that although the raw transmission speeds used in networks are increasing rapidly, the rate of advancement of microprocessor technology has slowed down. Therefore, network protocol-processing overheads have risen sharply in comparison with the time spent in packet transmission, resulting in degraded throughput for networked applications. More and more, it is the network end system, instead of the network, that is responsible for degraded performance of network applications. In this paper, the Linux system's packet receive process is studied from NIC to application. We develop a mathematical model to characterize the Linux packet receiving process. Key factors that affect Linux systems' network performance are analyzed.

Keywords: Linux, TCP/IP, protocol stack, process scheduling, performance analysis

1. Introduction

The computing models for High-Energy Physics (HEP) experiments are becoming ever more globally distributed and grid-based, both for technical reasons (e.g., to place computational and data resources near each other and the demand) and for strategic reasons (e.g., to leverage equipment investments). To support such computing models, the network and end systems, computing and storage, face unprecedented challenges. One of the biggest challenges is 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. Both the network and end systems should be able to provide the capabilities to support high

[*] Work supported by the U.S. Department of Energy under contract No. DE-AC02-76CH03000.
[+] Corresponding Author

systems should be able to provide the capabilities to support high bandwidth, sustained, end-to-end data transmission [1][2]. Recent trends in technology are showing that although the raw transmission speeds used in networks are increasing rapidly, the rate of advancement of microprocessor technology has slowed down [3][4]. Therefore, network protocol-processing overheads have risen sharply in comparison with the time spent in packet transmission, resulting in the degraded throughput for networked applications. More and more, it is the network end system, instead of the network, that is responsible for degraded performance of network applications.

Linux-based network end systems have been widely deployed in the HEP communities (e.g., CERN, DESY, Fermilab, SLAC). In Fermilab, thousands of network end systems are running Linux operating systems; these include computational farms, trigger processing farms, servers, and desktop workstations. From a network performance perspective, Linux represents an opportunity since it is amenable to optimization and tuning due to its open source support and projects such as web100 and net100 that enable tuning of network stack parameters [5][6]. In this paper, the Linux network end system's packet receive process is studied from NIC to application. We work with mature technologies rather than "bleeding-edge" hardware in order to focus on the end-system phenomena that stand between reasonable performance expectations and their fulfillment. Our analysis is based on Linux kernel 2.6.12. The network technology at layers 1 and 2 assumes an Ethernet medium, since it is the most widespread and representative LAN technology. Also, it is assumed that the Ethernet device driver makes use of Linux's "New API," or NAPI [7][8], which reduces the interrupt load on the CPUs. The contributions of the paper are as follows: (1) We systematically study the current packet handling in the Linux kernel. (2) We develop a mathematical model to characterize the Linux packet receive process. Key factors that affect Linux systems' network performance are analyzed. Through our mathematical analysis, we abstract and simplify the complicated kernel protocol processing into three stages, revolving around the ring buffer at the NIC driver level and sockets' receive buffer at the transport layer of the protocol stack. (3) Our experiments have confirmed and complemented our mathematical analysis.

The remainder of the paper is organized as follows: In Section 2 the Linux packet receiving process is presented. Section 3 presents a mathematical model to characterize the Linux packet receiving process. Key factors that affect Linux systems' network performance are analyzed. In Section 4, we show the experiment results that test and complement our model and further analyze the packet receiving process. Section 5 summarizes our conclusions.

2. Packet Receiving Process

Figure 1 demonstrates generally the trip of a packet from its ingress into a Linux end system to its final delivery to the application [7][9][10]. In general, the packet's trip can be classified into three stages:

- Packet is transferred from network interface card (NIC) to ring buffer. The NIC and device driver manage and control this process.

- Packet is transferred from ring buffer to a socket receive buffer, driven by a software interrupt request (softirq) [9][11][12]. The kernel protocol stack handles this stage.
- Packet data is copied from the socket receive buffer to the application, which we will term the *Data Receiving Process*.

In the following sections, we detail these three stages.

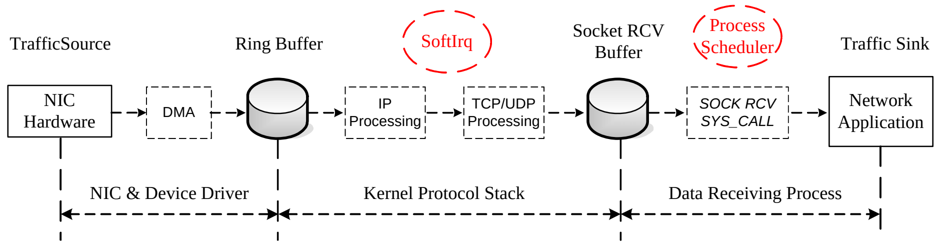

Figure 1 Linux Networking Subsystem: Packet Receiving Process

2.1 NIC and Device Driver Processing

The NIC and its device driver perform the layer 1 and 2 functions of the OSI 7-layer network model: packets are received and transformed from raw physical signals, and placed into system memory, ready for higher layer processing. The Linux kernel uses a structure *sk_buff* [7][9] to hold any single packet up to the MTU (Maximum Transfer Unit) of the network. The device driver maintains a “ring” of these packet buffers, known as a “ring buffer”, for packet reception (and a separate ring for transmission). A ring buffer consists of a device- and driver-dependent number of packet descriptors. To be able to receive a packet, a packet descriptor should be in “ready” state, which means it has been initialized and pre-allocated with an empty *sk_buff* which has been memory-mapped into address space accessible by the NIC over the system I/O bus. When a packet comes, one of the ready packet descriptors in the receive ring will be used, the packet will be transferred by DMA [13] into the pre-allocated *sk_buff*, and the descriptor will be marked as used. A used packet descriptor should be reinitialized and refilled with an empty *sk_buff* as soon as possible for further incoming packets. If a packet arrives and there is no ready packet descriptor in the receive ring, it will be discarded. Once a packet is transferred into the main memory, during subsequent processing in the network stack, the packet remains at the same kernel memory location.

Figure 2 shows a general packet receiving process at NIC and device driver level. When a packet is received, it is transferred into main memory and an interrupt is raised only after the packet is accessible to the kernel. When CPU responds to the interrupt, the driver’s *interrupt handler* is called, within which the

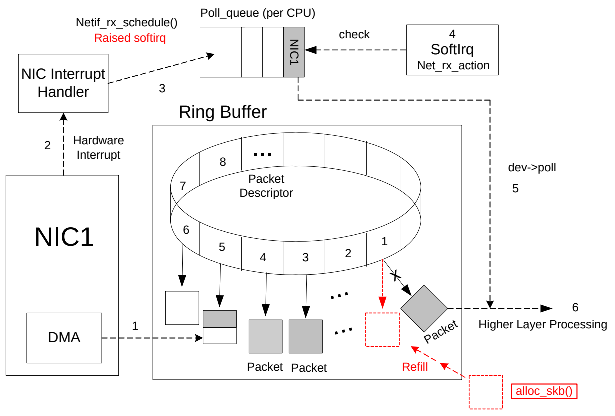

Figure 2 NIC & Device Driver Packet Receiving

softirq is scheduled. It puts a reference to the *device* into the *poll queue* of the interrupted CPU. The interrupt handler also disables the NIC's receive interrupt till the packets in its ring buffer are processed.

The softirq is serviced shortly afterward. The CPU polls each *device* in its *poll queue* to get the received packets from the ring buffer by calling the *poll* method of the *device driver*. Each received packet is passed upwards for further protocol processing. After a received packet is dequeued from its receive ring buffer for further processing, its corresponding packet descriptor in the receive ring buffer needs to be reinitialized and refilled.

2.2 Kernel Protocol Stack

2.2.1 IP Processing

The IP protocol receive function is called during the processing of a softirq for each IP packet that is dequeued from the ring buffer. This function performs initial checks on the IP packet, which mainly involve verifying its integrity, applying firewall rules, and disposing the packet for forwarding or local delivery to a higher level protocol. For each transport layer protocol, a corresponding handler function is defined: *tcp_v4_rcv()* and *udp_rcv()* are two examples.

2.2.2 TCP Processing

When a packet is handed upwards for TCP processing, the function *tcp_v4_rcv()* first performs the TCP header processing. Then the *socket* associated with the packet is determined, and the packet dropped if none exists. A socket has a lock structure to protect it from un-synchronized access. If the socket is locked, the packet waits on the backlog queue before being processed further. If the socket is not locked, and its *Data Receiving Process* is sleeping for data, the packet is added to the socket's *prequeue* and will be processed in batch in the *process context*, instead of the *interrupt context* [11][12]. Placing the first packet in the prequeue will wake up the sleeping data receiving process. If the prequeue mechanism does not accept the packet, which means that the socket is not locked and no process is waiting for input on it, the packet must be processed immediately by a call to *tcp_v4_do_rcv()*. The same function also is called to drain the backlog queue and prequeue. Those queues (except in the case of prequeue

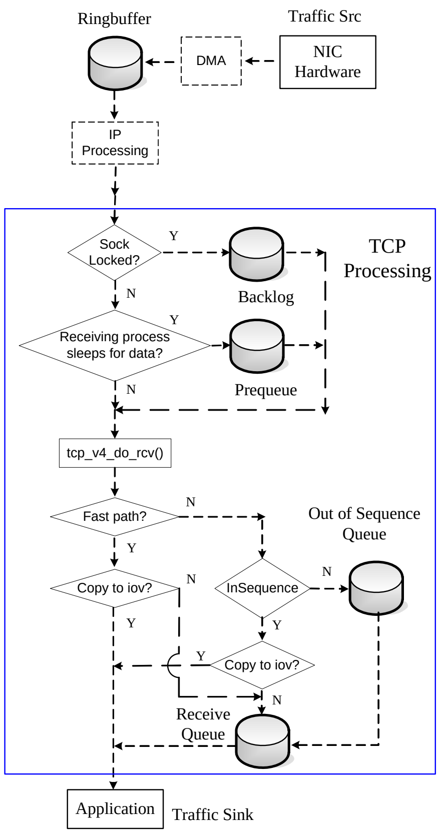

Figure 3 TCP Processing - Interrupt Context

overflow) are drained in the *process context*, not the *interrupt context* of the softirq. In the case of prequeue overflow, which means that packets within the prequeue reach/exceed the socket'*s* receive buffer quota, those packets should be processed as soon as possible, even in the *interrupt context*.

tcp_v4_do_rcv() in turn calls other functions for actual TCP processing, depending on the TCP state of the connection. If the connection is in *tcp_established* state, *tcp_rcv_established*() is called; otherwise, *tcp_rcv_state_process()* or other measures would be performed. *tcp_rcv_established()* performs key TCP actions: e.g. sequence number checking, DupACK sending, RTT estimation, ACKing, and data packet processing. Here, we focus on the data packet processing.

In *tcp_rcv_established(),* when a data packet is handled on the *fast path*, it will be checked whether it can be delivered to the user space directly, instead of being added to the *receive queue*. The data's destination in user space is indicated by an *iovec* structure provided to the kernel by the *data receiving process* through system calls such as *sys_recvmsg*. The conditions of checking whether to deliver the data packet to the user space are as follow:

- The socket belongs to the currently active process;
- The current packet is the next in sequence for the socket;
- The packet will entirely fit into the application-supplied memory location;

When a data packet is handled on the *slow path* it will be checked whether the data is in sequence (fills in the beginning of a hole in the received stream). Similar to the *fast path*, an in-sequence packet will be copied to user space if possible; otherwise, it is added to the *receive queue*. Out of sequence packets are added to the socket's *out-of-sequence queue* and an appropriate TCP response is scheduled. Unlike the *backlog queue, prequeue* and *out-of-sequence queue*, packets in the *receive queue* are guaranteed to be in order, already acked, and contain no holes. Packets in *out-of-sequence queue* would be moved to *receive queue* when incoming packets fill the preceding holes in the data stream. Figure 3 shows the TCP processing flow chart within the *interrupt context*.

As previously mentioned, the *backlog* and *prequeue* are generally drained in the *process context*. The socket's *data receiving process* obtains data from the socket through socket-related receive system calls. For TCP, all such system calls result in the final calling of *tcp_recvmsg(),* which is the top end of the TCP transport receive mechanism. As shown

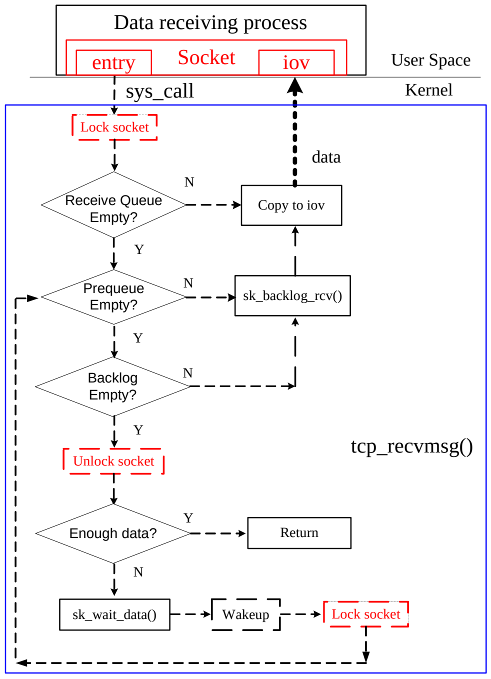

Figure 4 TCP Processing – Process Context

in Figure 4, when *tcp_recvmsg()* is called, it first locks the socket. Then it checks the *receive queue*. Since packets in the receive queue are guaranteed in order, acked, and without holes, data in *receive queue* is copied to user space directly. After that, *tcp_recvmsg()* will process the *prequeue* and *backlog queue*, respectively, if they are not empty. Both result in the calling of *tcp_v4_do_rcv()*. Afterward, processing similar to that in the *interrupt context* is performed. *tcp_recvmsg()* may need to fetch a certain amount of data before it returns to user code; if the required amount is not present, *sk_wait_data()* will be called to put the data receiving process to sleep, waiting for new data to come. The amount of data is set by the data receiving process. Before *tcp_recvmsg()* returns to user space or the data receiving process is put to sleep, the lock on the socket will be released. As shown in Figure 4, when the data receiving process wakes up from the sleep state, it needs to relock the socket again.

2.2.3 The UDP Processing

When a UDP packet arrives from the IP layer, it is passed on to *udp_rcv()*. *udp_rcv()'s* mission is to verify the integrity of the UDP packet and to queue one or more copies for delivery to multicast and broadcast sockets and exactly one copy to unicast sockets. When queuing the received packet in the *receive queue* of the matching socket, if there is insufficient space in the receive buffer quota of the socket, the packet may be discarded. Data within the socket's receive buffer are ready for delivery to the user space.

2.3 Data Receiving Process

Packet data is finally copied from the socket's receive buffer to user space by *data receiving process* through socket-related receive system calls. The receiving process supplies a memory address and number of bytes to be transferred, either in a *struct iovec*, or as two parameters gathered into such a struct by the kernel. As mentioned above, all the TCP socket-related receive system calls result in the final calling of *tcp_recvmsg()*, which will copy packet data from socket's buffers (*receive queue*, *prequeue*, *backlog queue*) through *iovec*. For UDP, all the socket-related receiving system calls result in the final calling of *udp_recvmsg()*. When *udp_recvmsg()* is called, data inside *receive queue* is copied through *iovec* to user space directly.

3. Performance Analysis

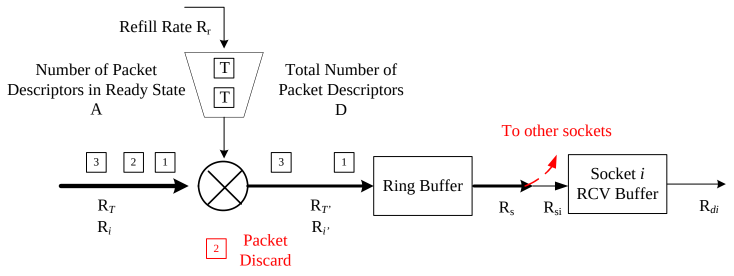

Figure 5 Packet Receiving Process - Mathematical Model

Based on the packet receiving process described in Section 2, the packet receiving process can be described by the model in Figure 5. In the mathematical model, the NIC and device driver receiving process can be represented by the token bucket algorithm [14], accepting a packet if a ready packet descriptor is available in the ring buffer and discarding it if not. The rest of the packet receiving processes are modeled as queuing processes [15].

We assume several incoming data streams are arriving and define the following symbols:

- $R_T(t)$, $R_{T'}(t)$: 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*;
- $\tau_{\min}$: The minimum time interval between a packet's ingress into the system and its first being serviced by a softirq;
- $R_{\max}$: NIC's maximum packet receive rate (Packets/Time Unit);
- $R_s(t)$: 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);
- N: The number of runnable processes during the packet reception period;
- $P_1, \ldots, P_N$: N runnable processes during the packet reception period;
- P_i: Data receiving process for data stream *i*;

The Token Bucket algorithm is a surprisingly good fit to the NIC and device driver receiving process. In our model, the receive ring buffer is represented as a 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.

Then, it has:

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

Therefore, to admit packets into the system without discarding, the system should meet the condition:

$$\forall t > 0, \qquad A(t) > 0 \qquad (2)$$

Also, it can be derived that:

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

It can be seen from (1) and (3) that in order to avoid or minimize packet drops by the NIC, the system needs either to raise its $R_r(t)$ and/or D, or to effectively decrease $R_T(t)$.

Since a used packet descriptor can be reinitialized and refilled after its corresponding packet is dequeued from the receive ring buffer for further processing, the rate of $R_r(t)$ depends on the following two factors: (1) Protocol packet service rate $R_s(t)$. To raise the protocol kernel packet service rate, approaches of optimizing or offloading the kernel packet processing in the system's protocol kernel have been proposed. For example, TCP/IP Offloading technology [16][17][18][19][20] aims to free the CPU of some packet processing by shifting tasks to the NIC or storage device. (2) The system memory allocation status. When the system is in memory pressure, allocation of new packet buffers is prone to failure. In that case, the used packet descriptor cannot be refilled; the rate of $R_r(t)$ is actually decreased. When all packet descriptors in the ready state are used up, further incoming packets will be dropped by the NIC. Experiments in Section 4.1 will confirm this point. In the absence of memory shortage, it can be assumed that $R_s(t) = R_r(t)$.

D is a design parameter for the NIC and driver. A larger D implies increased cost for the NIC. For a NAPI driver, D should be big enough to hold further incoming packets before the received packets in the NIC receive ring buffer are dequeued and the corresponding packet descriptors in the receive ring buffer are reinitialized and refilled. In that case, D should at least meet the following condition to avoid unnecessary packet drops:

$$D > \tau_{\min} * R_{\max} \quad (4)$$

Here, $\tau_{\min}$ is the minimum time interval between a packet's ingress into the system and its first being serviced by a softirq. In general, $\tau_{\min}$ includes the following components [12] [21]:

- NIC interrupt dispatch time (NIDT): when an NIC interrupt occurs, a system must save all registers and other system execution context before calling the NIC packet-receive interrupt service routine to handle it.
- NIC interrupt service time (NIST): the time used by the NIC interrupt service routine to retrieve information from the NIC and schedule the packet-receive softirq.

Among them, NIDT has nearly constant value given a hardware configuration. However, NIST depends on the length of the NIC interrupt handler. A poorly designed NIC device driver may impose a long NIST value and cause an unacceptably large $\tau_{\min}$. With a given D, a poorly designed NIC device driver can even cause packet drops in the receive ring buffer.

$R_T(t)$ is the offered total packet rate. Usually, one tries to increase $R_T(t)$ in order to maximize the incoming throughput. In order to avoid or minimize packet drops by the NIC, to decrease $R_T(t)$ seems to be an unacceptable approach. However, use of jumbo frames* [22][23][24] helps maintain the incoming byte rate while reducing $R_T(t)$ to avoid packet drops at the NIC. Using jumbo frames at 1Gb/s reduces the maximum packet rate from over 80,000 per second to under 14,000 per second. Since jumbo frames

* IEEE 802.3 Ethernet imposes a Maximum Transmission Unit (MTU) of 1500 bytes. But many Gigabit Ethernet vendors have followed a proposal by Alteon Networks to support Jumbo frame sizes over 9000 bytes.

decrease $R_{\max}$, it can be seen from (4) that the requirements for D might be lowered with jumbo frames.

The rest of the packet receiving processes are modeled as queuing processes. In the model, socket *i*'s receive buffer is a queue of size QB_i. The packets are put into the queue by softirq with a rate of $R_{si}(t)$, and are moved out of the queue by the *data receiving process* with a rate of $R_{di}(t)$.

For stream *i*, based on the packet receiving process, it has:

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

Similarity, it can be derived that:

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

In transport layer protocol operations $B_i(t)$ plays a key role. For UDP, when $B_i(t) \ge QB_i$, all the incoming packets for socket *i* will be discarded. In that case, all the protocol processing effort over the dropped packet would be wasted. From both the network end system and network application's perspective, this is the condition we try to avoid.

TCP does not drop packets at the socket level as UDP does when the receive buffer is full. Instead, it advertises $QB_i - B_i(t)$ to the sender to perform flow control. However, when a TCP socket's receive buffer is approaching full, the small window $QB_i - B_i(t)$ advertised to the sender side will throttle the sender's data sending rate, resulting in degraded TCP transmission performance [25].

From both UDP and TCP's perspectives, it is desirable to raise the value of $QB_i - B_i(t)$, which is:

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

Clearly it is not desirable to reduce $R_{si}(t)$ to achieve the goal. But the goal can be achieved by raising QB_i and/or $R_{di}(t)$. For most operating systems, QB_i is configurable. For Linux 2.6, QB_i is configurable through /proc/net/ipv4/tcp_rmem, which is an array of three elements, giving the minimum, default, and maximum values for the size of the receive buffer.

To maximize TCP throughput, the rule of thumb for configuring TCP QB_i is to set it to the Bandwidth*Delay Product (BDP) of the end-to-end path (the TCP send socket buffer size is set the same way). Here Bandwidth means the available bandwidth of the end-to-end path; and Delay is the round trip time. According to the above rules, for long, high-bandwidth connections, QB_i would be set high. IA-32 Linux systems usually adopt a 3G/1G virtual address layout, 3GB virtual memory for user space and 1GB for kernel space [12][26][27]. Due to this virtual address partition scheme, the kernel can at most directly map 896MB physical memory into its kernel space. This part of memory is called Lowmem. The kernel code and its data structures must reside in Lowmem, and they are not swappable. However, the memory allocated for $QB_i s$ (and the send socket buffers) also has to reside within Lowmem, and is also not swappable. In that case, if QB_i is set high (say, 5MB or 10MB) and the system has a large number of TCP connections (say, hundreds), it will soon run out of Lowmem. "Bad things happen when you're out of Lowmem" [12][26][27]. For example, one direct consequence is to cause packet drops at the NIC: due to memory shortage in Lowmem, the used packet descriptor cannot be refilled; when all packet descriptors in the ready state are used up, further incoming packets will be dropped at the NIC. To prevent TCP from overusing the system memory in Lowmem, the Linux TCP implementation has a control variable - *sysctl_tcp_mem* to bound the total amount of memory used by TCP for the entire system. *Sysctl_tcp_mem* is configurable through /proc/net/ipv4/tcp_mem, which is an array of three elements, giving the *minimum*, *memory pressure point*, and *high* number of pages allowed for queuing by all TCP sockets. For IA-32 Linux systems, if QB_i would be set high, the *sysctl_tcp_mem* should be correspondingly configured to prevent system from running out of Lowmem. For IA-64 Linux systems Lowmem is not so limited, and all installed physical memory belongs to Lowmem. But configuring QB_i and *sysctl_tcp_mem* is still subject to physical memory limit.

$R_{di}(t)$ is contingent on the data receiving process itself and the offered system load. The offered system load includes the offered interrupt load and offered process load. Here, the offered interrupt load means all the interrupt-related processing and handling (e.g., NIC interrupt handler processing, packet receiving softirq processing). In an interrupt-driven operating system like Linux, since interrupts have higher priority than processes, when the offered interrupt load exceeds some threshold, the user-level processes could be starved for CPU cycles, resulting in decreased $R_{di}(t)$. In the extreme, when the user-level processes were totally preempted by interrupts, $R_{di}(t)$ would drop to zero. For example, in the condition of receive livelock [8], when non-flow-controlled packets arrive too fast, the system will spend all of its time processing receiver interrupts. It will therefore have no CPU resources left to support delivery of the arriving packets to data receiving process, with $R_{di}(t)$ dropping to zero. With heavy network loads, the following approaches are usually taken to reduce the offered interrupt load and save CPU cycles for network applications: (1) Interrupt coalescing (NAPI) [8], which reduces the cost of packet receive interrupts by processing several packets for each interrupt. (2) Jumbo frames [22][23][24]. As stated above, jumbo frames can effectively reduce the incoming packet rate, hence the interrupt rate and the interrupt load. Specifically, jumbo frames will reduce network stack processing (softirq processing) overhead incurred per byte. A significant reduction of CPU utilization can be obtained. (3) TCP/IP offloading [17][18].

In the following sections, we discuss $R_{di}(t)$ from the offered process load's perspective, assuming the system is not overloaded.

Linux 2.6 is a *preemptive multi-processing* operating system. Processes (tasks) are scheduled to run in a way of *prioritized round robin* [11][12][28]. As shown in Figure 6, the whole process scheduling is based on a data structure called *runqueue*. Essentially, a runqueue keeps track of all runnable tasks assigned to a particular CPU. As such, one runqueue is created and maintained for each CPU in a system. Each runqueue contains two priority arrays: *active priority array* and *expired priority array*. Each priority array contains one queue of runnable processes per priority level. Higher priority processes are scheduled to run first. Within a given priority, processes are scheduled round robin. All tasks on a CPU begin in the active priority array. Each process' *time slice* is calculated based on its *nice* value; as a process in the active priority array run out of its time slice, the process is considered expired. An expired process is moved to the expired priority array[+]. During the move, a new time slice and priority is calculated. When there are no more runnable tasks in the active priority array, it is simply swapped with the expired priority array.

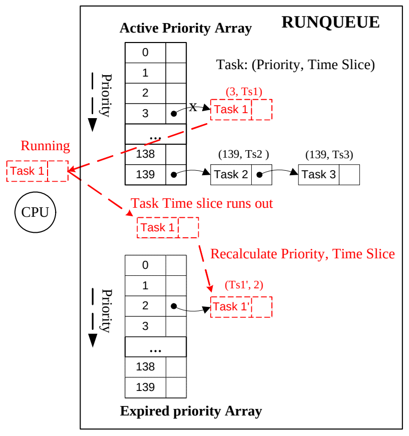

Figure 6 Linux Process Scheduling

Let's assume that during the period of data reception, the system process load is stable. There are N running processes: P_1, ... , P_N; and P_i is the data receiving process for data stream i. P_i's packet service rate is constant λ when the process is running. Each process will not sleep (e.g., waiting for I/O) or a sleeping process is waking up soon compared with its time slice. As such, data receiving process P_i's running model can be modeled as shown in Figure 7. Interrupts might happen when a process runs. Since interrupt process time is not charged upon processes, we do not consider interrupts here.

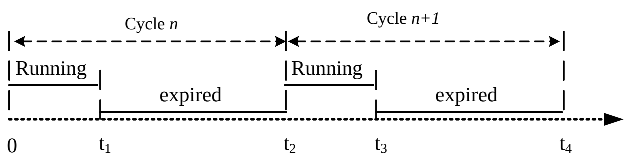

Figure 7 Data receiving process running model

Further, the running cycle of P_i can be derived as follows:

$$\sum_{j=1}^{N} Timeslice(P_j) \tag{8}$$

In the model, process P_i's running period is $Timeslice(P_i)$, and the expired period is:

$$\sum_{j=1}^{N} Timeslice(P_j) - timeslice(P_i) \tag{9}$$

[+] To improve system responsiveness, an interactive process is moved back to the active priority array.

From a process' perspective, process P_i 's relative CPU share is:

$$Timeslice(P_i) / \sum_{j=1}^{N} Timeslice(P_j) \qquad (10)$$

From (8), (9), and (10), it can be seen that when process' nice values are relatively fixed (e.g., in Linux, the default nice value is 0), the number of N will dictate P_i 's running frequency.

For the cycle *n* in Figure 7, we have:

$$R_{di}(t) = \begin{cases} \lambda, & 0 < t < t_1 \\ 0, & t_1 < t < t_2 \end{cases} \qquad (11)$$

Therefore, to raise the rate of $R_{di}(t)$ requires increasing the data receiving process' CPU share: either increase data receiving process' time slice/nice value, or reduce the system load by decreasing N to increase data receiving process' running frequency. Experiment results in Section 4.3 will confirm this point.

Another approach to raise the rate of $R_{di}(t)$ is to increase the packet service rate λ. From a programmers' perspective, the following optimizations could be taken to maximize λ: (1) Buffer alignments [29][30]; (2) Asynchronous I/O [30].

4. Results and Analysis

We run the data transmission experiments upon Fermi's sub-networks. In the experiments, we run *iperf* [31] to send data in one direction between two computer systems. *iperf* in the receiver is the data receiving process. As shown in Figure 8, the sender and the receiver are connected to two Cisco 6509 switches respectively. The corresponding connection's bandwidth is as labeled. The sender and receiver's features are as shown in table 1.

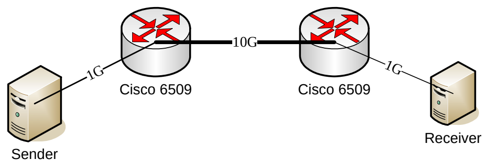

Figure 8 Experiment Network & Topology

	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
OS	Linux 2.6.12 (3G/1G virtual address layout)	Linux 2.6.12 (3G/1G virtual address layout)

Table 1 Sender and Receiver Features

[†] We ran experiments on different versions of Linux receivers, and similar results were obtained.

In order to study the detailed packet receiving process, we have added instrumentation within the Linux packet receiving path. Also, to study the system's reception performance at various system loads, we are compiling the Linux Kernel as background system load by running `make –nj` [11]. The different value of *n* corresponds to different levels of background system load, e.g. `make –4j`. For simplicity, they are termed as "BL*n*". The background system load implies load on both CPU and system memory.

We run *ping* to obtain the round trip time between Sender and Receiver. The maximum RTT is around 0.5ms. The BDP of the end-to-end path is around 625KB. When TCP sockets' receive buffer sizes are configured larger than BDP, the TCP performance won't be limited by the TCP flow control mechanism (Small TCP sockets' receive buffer size would limit the end-to-end performance, readers could refer to [25][32]). To verify this point, we run experiments with various receiver buffer sizes equal or greater than 1MB: sender transmits one TCP stream to receiver for 100 seconds, all the processes run with a nice value of 0. The experiment results are as shown in Table 2. It can be seen that: when the TCP sockets' receive buffer sizes are greater than BDP, similar results (End-to-End Throughputs) have been obtained.

	Experiment Results: End-to-End Throughput				
Receive Buffer Size	**1M**	**10M**	**20M**	**40M**	**80M**
BL0	310 Mbps	309 Mbps	310 Mbps	309 Mbps	308 Mbps
BL4	64.7 Mbps	63.7 Mbps	63.9 Mbps	65.2 Mbps	65.5 Mbps
BL10	30.7 Mbps	31 Mbps	31.4 Mbps	31.9 Mbps	31.9 Mbps

Table 2 TCP Throughput with various Socket Receive Buffer Sizes

In the following experiments, unless otherwise specified, all the processes are running with a nice value of 0; and *iperf's* receive buffer is set to 40MB. From the system level, the *sysctl_tcp_mem* is configured as: "49152 65536 98304". We choose a relatively large receiver buffer based on the following considerations: (1) In the real world, system administrators often configure /proc/net/ipv4/tcp_rmem high to accommodate high BDP connections. (2) We want to demonstrate the potential dangers brought to the Linux systems when configuring /proc/net/ipv4/tcp_rmem high.

4.1 Receive Ring Buffer

The total number of packet descriptors in the receive ring buffer of the receiver's NIC is 384. As it has been put in section 3, the receive ring buffer might be a potential bottleneck for packet receiving. Our experiments have confirmed this point. In the experiments, Sender transmits 1 TCP stream to Receiver with the transmission duration of 25 seconds. The experiment results are as shown in Figure 9:

- Normally, only a small portion of the packet descriptors in the receive ring buffer are used, and the used descriptors are reinitialized and refilled in time.
- Surprisingly it can be seen that on a few occasions (@2.25s, @2.62s, @2.72s) with the load of BL*10*, all 384 packet descriptors in the receive ring buffer were used. At these times further incoming packets are dropped until the used descriptors are reinitialized and refilled. Upon careful investigation, it was determined that with BL*10,* the system is in high memory pressure. In this condition attempts to allocate new packet buffers to refill used descriptors fail and the rate of $R_r(t)$ is

actually decreased; soon the receive ring buffer runs out of ready descriptors and packets are dropped. Those failed-to-be-refilled used descriptors can only be refilled after the Page Frame Reclaiming Algorithm (PFRA) of the Linux kernel refills the lists of free blocks of the buddy system, for example, by shrinking cache or by reclaiming page frames from User Mode processes [12].

In the real world, packet loss is generally blamed on the network, especially for TCP traffic. Few people are conscious that packets drops might commonly occur at the NIC.

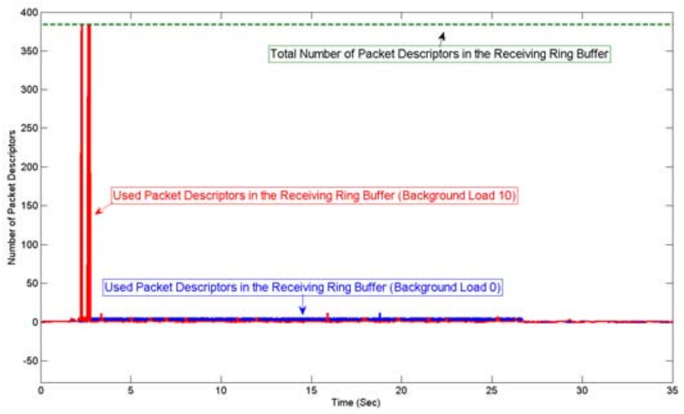

Figure 9 Used Packet Descriptors in the Receiving Ring Buffer

4.2 TCP & UDP

In the TCP experiments, sender transmits one TCP stream to receiver for 25 seconds. Figures 10 and 11 show observations at background loads (BL) of 0 and 10 respectively.

- Normally, prequeue and out-of-sequence queue are empty. The backlog queue is usually not empty. Packets are not dropped or reordered in the test network. However, when packets are dropped by the NIC (Figure 9) or temporarily stored in the backlog queue, subsequent packets may go to the out-of-sequence queue.
- The receive queue is approaching full. In our experiment, since the sender is more powerful than the receiver, the receiver controls the flow rate. The experiment results have confirmed this point.
- In contrast with Figure 10, the backlog and receive queues in Figure 11 show some kind of periodicity. The periodicity matches the data receiving process' running cycle. In Figure 10, with BL0, the data receiving process runs almost continuously, but at BL10, it runs in a prioritized round-robin manner.

In the UDP experiments, sender transmits one UDP stream to receiver for 25 seconds. 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. Figures 12 and 13 show the results for UDP transmissions.

- Both cases (1) and (2) are within receiver's handling limit. The receive buffer is generally empty.

- In case (3), the receive buffer remains full. Case (3) exhibits receive-livelock problems [8]. Packets are dropped in the socket level. The effective data rate in case (3) is 88.1Mbits, with a packet drop rate of 670612/862066 (78%) at the socket.

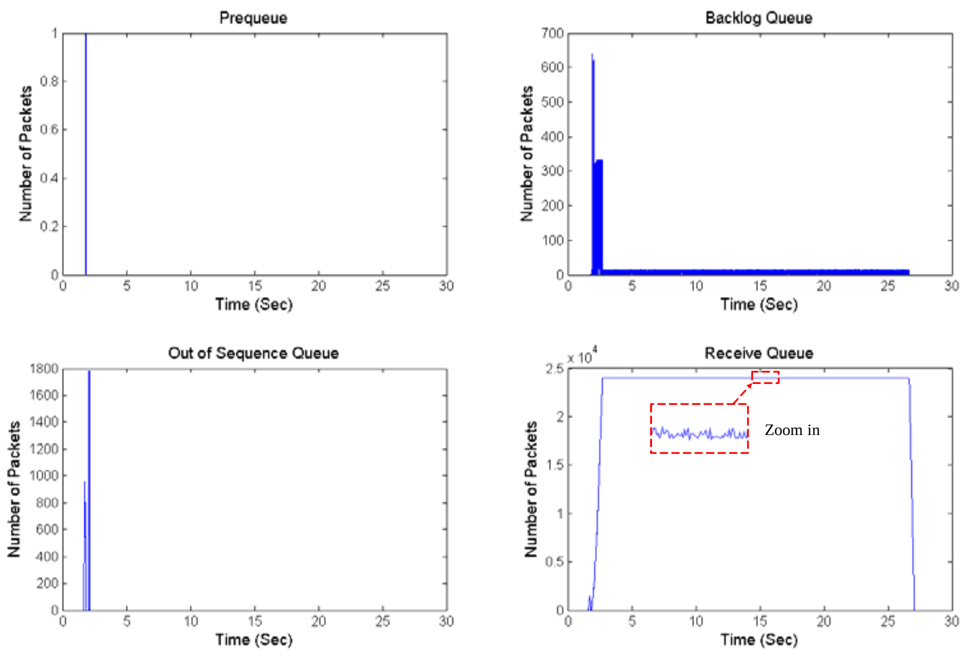

Figure 10 Various TCP Receive Buffer Queues – Background Load 0

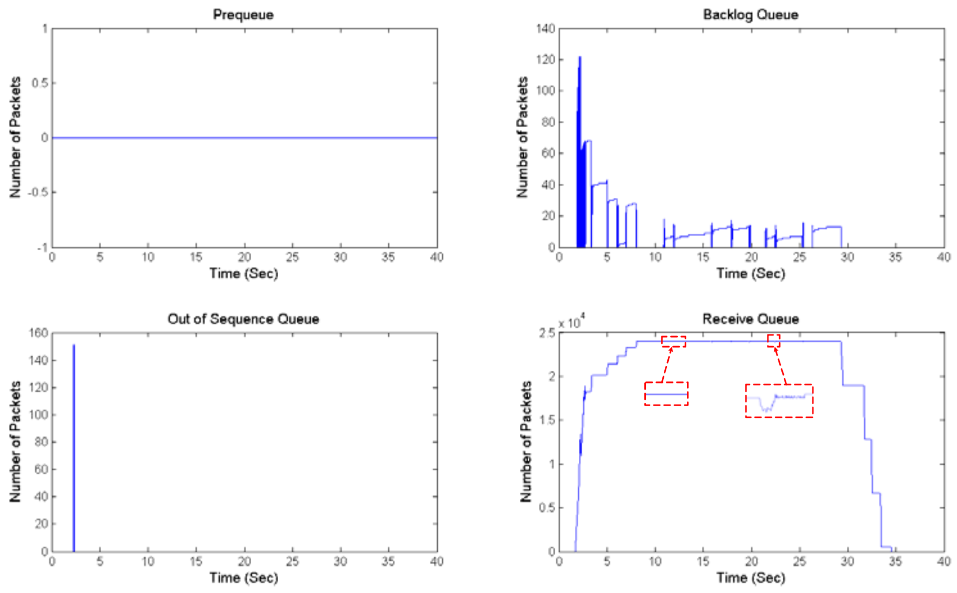

Figure 11 Various TCP Receive Buffer Queues – Background Load 10

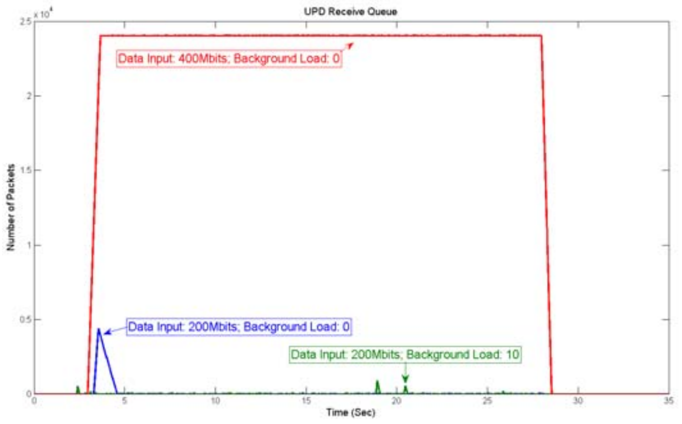

Figure 12 UDP Receive Buffer Queues at various conditions

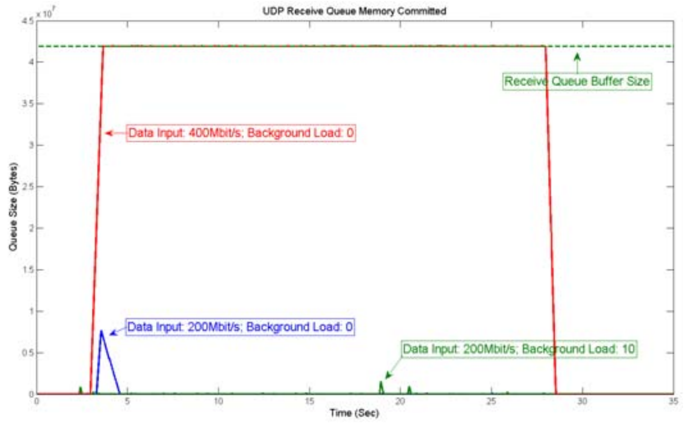

Figure 13 UDP Receive Buffer Committed Memory

The above experiments have shown that when the sender is faster than the receiver, TCP (or UDP) receiver buffers are approaching full. When the socket receive buffer size is set high, a lot of memory will be occupied by the full receive buffers (the same for the socket send buffer, which is beyond the topic of this paper). To verify this point, we run the experiment as follows: sender transmits one TCP stream to receiver for 100 seconds, all the processes run with a nice value of 0. We record the Linux system's *MemFree*, *Buffers*, *Cached* as shown in /proc/meminfo at the 50 seconds point (based on the above experiments, the receive buffer is approaching full at 50s). Here, *MemFree* is the size of the total available free memory in the system. *Buffers* and *Cached* are the sizes of the in-memory *buffer cache* and *page cache* respectively. When the system is in memory pressure, the page frames allocated for the buffer cache and the page cache will be reclaimed by the PFRA [12][27]. Also, please note that: (1) since the total system memory is 256MB, all of them belong to Lowmem. (2) *sysctl_tcp_mem* is configured as "49152 65536 98304", which means that the maximum TCP memory is allowed to reach as high as 384MB‡, if possible.

The experiment results are as shown in Table 3. It can be seen that with increased socket receive buffer size, the system's free memory clearly decreases. Specifically, when the

‡ A page is 4K.

socket receive buffer size is set as 170MB, both the buffer cache and the page cache are shrunk. The page frames allocated for the buffer cache and the page cache are reclaimed by PFRA to save memory for TCP. It can be contemplated that if the socket receive buffer is set high and there are multiple simultaneous connections, the system can easily run out of memory. When the memory is below some threshold and the PFRA cannot reclaim page frames any more, the out of memory (OOM) killer [12] starts to work and selects processes to kill to free page frames. To the extreme, the system might even crash. To verify this point, we set the socket receive buffer size to 80MB, and run five TCP connections simultaneously to the receiver, the receiver soon ran out of memory and killed the iperf process.

	Experiment Results					
Receive Buffer Size	**1M**	**10M**	**40M**	**80M**	**160M**	**170M**
MemFree (KB)	200764	189108	149056	95612	3688	3440
Buffers (KB)	7300	7316	7384	7400	7448	2832
Cached (KB)	28060	28044	28112	28096	14444	6756

Table 3 Linux System's Free Memory with Various Receive Buffer Size

Clearly, for a system with 256MB memory, allowing the overall TCP memory to reach as high as 384MB is wrong. To fix the problem, we reconfigure the *sysctl_tcp_mem* as "40960 40960 40960" (the maximum TCP memory is allowed to reach at most 160MB). Again, we set the socket receive buffer size to 80MB, and repeat the above experiments. No matter how many TCP connections are simultaneously streamed to the receiver, both iperf process and the receiver work well.

The above experiments have shown that when /proc/net/ipv4/tcp_rmem (or /proc/net/ipv4/tcp_wmem) is set high, the /proc/net/ipv4/tcp_mem should be correspondingly configured to prevent system from running out of Lowmem. For IA-32 architecture Linux network systems with memory larger than 1G, we suggest to limit the overall TCP memory to 600MB at most. As said in Section 3, this is due to the facts that: the IA-32 architecture Linux network systems usually adopt the 3G/1G virtual address layout, and the kennel can at most have 896MB of Lowmem; the kernel code and its data structures must reside in Lowmem, and they are not swappable; the memory allocated for socket receive buffers (and send buffers) also have to reside within Lowmem, and they are also not swappable. When the socket receive buffer size is set high and there are multiple simultaneous connections, the system can easily run out of Lowmem. The overall TCP memory must be limited.

4.3 Data receiving process

The object of the next experiment is to study the overall receiving performance when the data receiving process' CPU share is varied. In the experiments, 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.

A Linux process' nice value (static priority) ranges from –20 to +19 with a default of zero. Nineteen is the lowest and –20 is the highest priority. The nice value is not changed by the kernel. A Linux process' time slice is calculated purely based on its nice value. The higher a process' priority, the more time slice it receives per round of execution, which implies a greater CPU share. Table 4 shows the time slices for various nice values.

Nice value	Time slice
+19	5 *ms*
0	100 *ms*
-10	600 *ms*
-15	700 *ms*
-20	800 *ms*

Table 4 Nice value vs. Time slice

The experiment results in Figure 14 have shown:

- The higher a data receiving process' priority, the more CPU time it receives per round of execution. The higher CPU share entails the relative higher actual packet service rate from the socket's receive buffer, resulting in improved end-to-end data transmission.
- The greater the background load, the longer each complete round of execution takes. This reduces the running frequency of data receiving process and its overall CPU share. When the data receiving process is less scheduled to run, data inside the receive buffer is less frequently serviced. Then TCP flow control mechanism will take effect to throttle the sending rate, resulting in degraded end-to-end data transmission rate.

Experiment results confirm and complement our mathematical analysis in section 3.

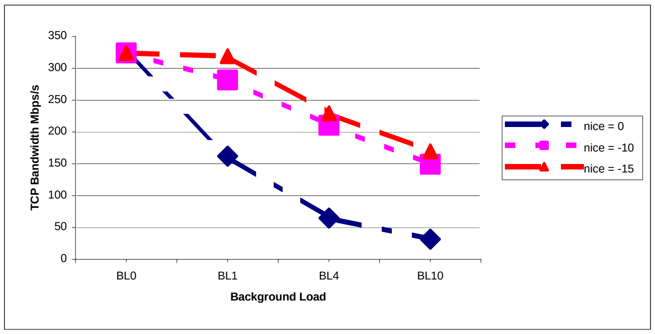

Figure 14 TCP Data Rate at Various Conditions

5. Conclusion

In this paper, the Linux system's packet receive process is studied in detail from NIC to application. We develop a mathematical model to characterize and analyze the Linux packet receive process. In the mathematical model, the NIC and device driver receiving process is represented by the token bucket algorithm; and the rest of the packet receiving processes are modeled as queuing processes.

Experiments and analysis have shown that incoming packets might be dropped by the NIC when there is no ready packet descriptor in the receive ring buffer. In overloaded systems, memory pressure usually is the main reason that causes packet drops at the NIC: due to memory shortage in Lowmem, the used packet descriptor cannot be refilled; when all packet descriptors in the ready state are used up, further incoming packets will be dropped by the NIC.

Experiments and analysis have also shown that the data receiving process' CPU share is another influential factor for the network application's performance. Before consumed by the data receiving process, the received packets are put into sockets' receive buffers. For UDP, when a socket's receive buffer is full, all the incoming packets for the socket will be discarded. In that case, all the protocol processing effort over the dropped packet would be wasted. For TCP, a full receive buffer will throttle the sender's data sending rate, resulting in degraded TCP transmission performance. To raise the data receiving process' CPU share, the following approaches could be taken: interrupt coalescing, jumbo frames, TCP/IP offloading, reducing the offered system load, lowering the data receiving process' nice value etc.

For the IA-32 architecture Linux network systems, more attention should be paid when enabling big socket receive (send) buffer size: configuring /proc/net/ipv4/tcp_rmem (or /proc/net/ipv4/tcp_wmem) high is good to the performance of high BDP connections, but the system might run out of Lowmem. Therefore, the /proc/net/ipv4/tcp_mem should be correspondingly configured to prevent system from running out of Lowmem.

We studied systems with network and CPU speeds that are moderate by today's standards in order to deal with mature overall system design. We expect our results to hold as all parts of the system scale up in speed, until and unless some fundamental changes are made to the packet receiving process.

References

[1] H. Newman, *et al.*, "The Ultralight Project: the Network as an Integrated and Managed Resource for Data-intensive Science". Computing in Science & Engineering, vol. 7, no. 6, pp. 38 – 47.

[2] A. Sim *et al.*, "DataMover: Robust Terabyte-scale Multi-file Replication over Wide-area Networks," Proceedings of 16th International Conference on Scientific and Statistical Database Management, 2004, pp. 403 – 412.

[3] D. Matzke, "Will Physical Scalability Sabotage Performance Gains?" Computer, vol. 30, no. 9, Sep 1997, pp. 37 – 39.

[4] D. Geer, "Chip makers turn to multicore processors," Computer, vol. 38, no. 5, May 2005, pp. 11 – 13.

[5] M. Mathis *et al.*, "Web100: Extended TCP Instrumentation for Research, Education and Diagnosis," ACM Computer Communications Review, vol. 33, no. 3, July 2003.

[6] T. Dunigan *et al.*, "A TCP Tuning Daemon," SuperComputing 2002.

[7] M. Rio *et al.*, "A Map of the Networking Code in Linux Kernel 2.4.20," March 2004.

[8] J. C. Mogul *et al.*, "Eliminating Receive Livelock in an Interrupt-driven Kernel," ACM Transactions on Computer Systems, vol. 15, no. 3, pp. 217--252, 1997.

[9] K. Wehrle *et al.*, The Linux Networking Architecture – Design and Implementation of Network Protocols in the Linux Kernel, Prentice-Hall, ISBN 0-13-177720-3, 2005.

[10] www.kernel.org.

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

[12] D. P. Bovet, Understanding the Linux Kernel, 3rd Edition, O'Reilly Press, ISBN: 0-596-00565-2, 2005.

[13] J. Corbet *et al.*, Linux Device Drivers, 3rd Edition, O'Reilly Press, ISBN: 0-596-00590-3, 2005.

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

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

[16] Y. Hoskote *et al.*, "A TCP offload accelerator for 10 Gb/s Ethernet in 90-nm CMOS," IEEE Journal of Solid-State Circuits, vol. 38, no. 11, Nov. 2003, pp. 1866 – 1875.

[17] G. Regnier *et al.*, "TCP Onloading for Data Center Servers," Computer, vol. 37, no. 11, Nov. 2004, pp. 48 – 58.

[18] D. Freimuth *et al.*, "Server Network Scalability and TCP Offload," Proceedings of the 2005 USENIX Annual Technical Conference, pp. 209--222, Anaheim, CA, Apr. 2005.

[19] J. Mogul, "TCP Offload is a Dumb Idea Whose Time has Come," Proceedings of the 9th Workshop on Hot Topics in Operating Systems, Lihue, Hawaii, May 2003.

[20] D. D. Clark *et al.*, "An Analysis of TCP Processing Overheads," IEEE Communication Magazine, vol. 27, no. 2, June 1989, pp. 23 – 29.
[21] K. Lin *et al.*, "The Design and Implementation of Real-time Schedulers in RED-Linux," Proceedings of the IEEE, vol. 91, no. 7, pp. 1114-1130, July 2003.
[22] S. Makineni *et al.*, “Performance Characterization of TCP/IP Packet Processing in Commercial Server Workloads,” IEEE International Workshop on Workload Characterization, Oct. 2003, pp. 33 – 41.
[23] Y. Yasu *et. al.*, “Quality of Service on Gigabit Ethernet for Event Builder,” IEEE Nuclear Science Symposium Conference Record, 2000, vol. 3, pp. 26/40 - 26/44.
[24] J. Silvestre *et al.*, “Impact of the Use of Large Frame Sizes in Fieldbuses for Multimedia Applications,” 10th IEEE Conference on Emerging Technologies and Factory Automation, vol. 1, pp. 433-440, 2005.
[25] Transmission Control Protocol, RFC 793, 1981.
[26] www.linux-mm.org.
[27] M. Gorman, Understanding the Linux Virtual Memory Manager, Prentice Hall PTR, ISBN: 0131453483, April 2004.
[28] C. S. Rodriguez *et al.*, The Linux(R) Kernel Primer: A Top-Down Approach for x86 and PowerPC Architectures, Prentice Hall PTR, ISBN: 0131181637, 2005.
[29] L. Arber *et al.*, “The Impact of Message-buffer Alignment on Communication Performance,” Parallel Processing Letters, vol. 15, no. 1-2, 2005, pp. 49-66.
[30] K. Chen *et al.*, “Improving Enterprise Database Performance on Intel Itanium Architecture,” Proceedings of the Linux Symposium, July 23-26, 2003, Ottawa, Ontario, Canada.
[31] http://dast.nlanr.net/Projects/Iperf/.
[32] J. Semke *et al.*, “Automatic TCP Buffer Tuning,” Computer Communication Review, ACM SIGCOMM, vol. 28, no. 4, Oct. 1998.