View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UNT Digital Library

FERMILAB-PUB-06-432-CD

Potential Performance Bottleneck in Linux TCP

Wenji Wu*, Matt Crawford*
Fermilab, MS-368, P.O. Box 500
Batavia, IL, 60510
wenji@fnal.gov, crawdad@fnal.gov

Abstract

TCP is the most widely used transport protocol on the Internet today. Over the years, especially recently, due to requirements of high bandwidth transmission, various approaches have been proposed to improve TCP performance. The Linux 2.6 kernel is now preemptible. It can be interrupted mid-task, making the system more responsive and interactive. However, we have noticed that Linux kernel preemption can interact badly with the performance of the networking subsystem. In this paper we investigate the performance bottleneck in Linux TCP. We systematically describe the trip of a TCP packet from its ingress into a Linux network end system to its final delivery to the application; we study the performance bottleneck in Linux TCP through mathematical modeling and practical experiments; finally we propose and test one possible solution to resolve this performance bottleneck in Linux TCP.

Keywords: Linux, TCP, Networking, Process scheduling, Performance analysis, Protocol stack

1. Introduction

The Transmission Control Protocol (TCP) is the most widely used transport protocol on the Internet today. It carries the vast majority of all traffic over the Internet for various network applications, including end-user applications (such as web browsing, remote login, and email), bandwidth-intensive applications (such as GridFTP for bulk data transmission [1]) and high-performance distributed computing [2][3]. TCP has been and will continue to be an evolving protocol. Over the years, various TCP flavors have been implemented. Early TCP implementations used a go-back-N model and required the expiration of a retransmission timer to recover any loss. TCP Tahoe added the slow start, congestion avoidance, and fast retransmit algorithms to TCP [4]. Based on TCP Tahoe, TCP Reno added fast recovery algorithm, first implemented in 1990 [5]. TCP SACK allows receivers to selective ACK out of sequence data and it aimed at eliminating the timeouts that arise in TCP Reno due to multiple losses from the same window [6][7]. TCP Vegas [8] is another implementation of TCP, which adjusts transmission rate according to anticipated congestion. It employs a new retransmission mechanism and slow start mechanism from Reno. TCP Westwood [9][10] is proposed to handle random or sporadic losses. It continuously measures at the TCP source the rate of the connection by monitoring the rate of returning ACKs. The estimate is then used to compute congestion window and slow start threshold after a congestion episode. Recently, due to requirements for high bandwidth transmission, TCP variants, such as FAST TCP [11], BIC TCP [12], HTCP [13], and HSTCP [14], are also proposed and implemented.

* Work supported by the U.S. Department of Energy under contract No. DE-AC02-76CH03000.

To improve TCP performance, researchers have been primarily working in the fields of TCP flow control [15], TCP congestion control [4 - 14][16][17][18], and TCP offloading [19][20].

Linux-based network end systems have been widely deployed in the High Energy Physics (HEP) community, at laboratories and universities. At 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 due to its open source support and projects such as web100 and net100 that enable tuning of network stack parameters [21][22].

In all pervious versions of Linux the kernel itself cannot be interrupted while it is processing. Linux 2.6 is preemptible [27][29]. The 2.6 kernel can be interrupted mid-task, so that the system is more responsive and interactive. However, we notice that preemption in certain sections incurs some serious negative effects on networking performance. In this paper, we investigate these problems. Our analysis is based on Linux kernel 2.6.14. The contribution of the paper is as follows: (1) we systematically describe the trip of a TCP packet from its ingress into a Linux end system to its final delivery to the application; (2) we point out the performance bottleneck in Linux TCP from both mathematical analysis and practical experiments; (3) we propose and test one possible solution to resolve the performance bottleneck in Linux TCP.

The remainder of the paper is organized as follows: In Section 2 the Linux packet receiving process is presented. Section 3 analyzes the performance bottleneck in Linux TCP. In Section 4, we show the experiment results to further study the Linux TCP performance issues, verifying our conclusions in Section 3. In Section 5, we propose a potential solution to resolve the performance bottleneck in Linux TCP. And finally in section 6, we conclude the paper.

2. TCP Packet Receiving Process

The Layer 2 technology is assumed Ethernet network media, 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 [23][24], which reduces the interrupt load on the CPUs.

Figure 1 demonstrates generally the trip of a TCP packet from its ingress into a Linux end system to its final delivery to the application [23][25][26]. We will not generally observe the distinctions among datalink frames, IP packets, and TCP segments, as the data structures moved along protocol stack in the Linux kernel represent any of these things at different times, and the data remains at the same memory location. For simplicity, we use the single term "packet" wherever it will not cause confusion. 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 controls this process.

- Packet is transferred from ring buffer to a socket receive buffer, driven by a software interrupt request (softirq) [25][27]. 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*.

The following subsections detail these three stages.

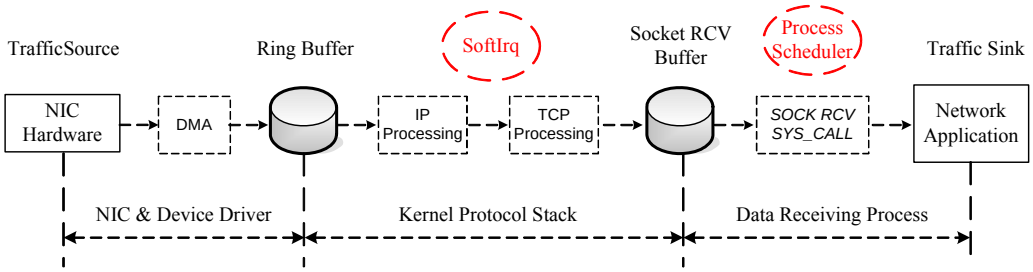

Figure 1 Linux Networking Subsystem: TCP Packets Receiving Process

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 (datalink frames) 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* [23][25] 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* that 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 reception ring will be used, the packet will be transferred by DMA [28] 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 reception 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.

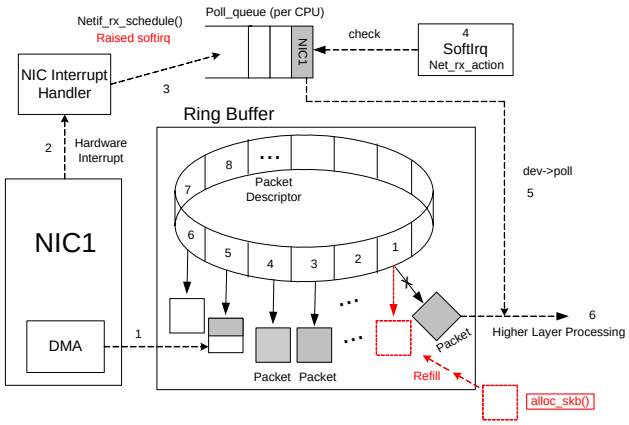

Figure 2 NIC & Device Driver Packet Receiving

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 *softirq* is scheduled by calling *netif_rx_schedule()*. 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 until all 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 *dev->poll()* of the *device driver*. In *dev->poll(),* each received packet is passed upwards for further processing by *net_receive_skb()*. After a received packet is dequeued from its receiving ring buffer for further processing, its corresponding packet descriptor in the reception ring buffer needs to be reinitialized and refilled.

2.2 Kernel Protocol Stack

2.2.1 IP Processing

The IP protocol receive function *ip_rcv()* gets called from within *net_receive_skb()* during the processing of a softirq, whenever an IP packet is dequeued from its receiving ring buffer. This function performs all the initial checks on the IP packet, which mainly involve verifying its integrity (IP checksum, IP header fields and minimum packet length). If the packet looks correct and passes the *netfilter* hook, *ip_rcv_finish()* is called. *ip_rcv_finish()* deals with the routing functionality of IP. It checks whether the packet should be forwarded to another machine or if it is destined to the local host. In the latter case, the packet is given to *ip_local_deliver().* In case the packet is fragmented, IP fragment reassembly is performed here. Then, the packet passes another *netfilter* hook, and finally goes to the *ip_local_deliver_finish()* function. There, an IP packet undergoes the last stage of IP-level processing: IP header data is trimmed and the higher layer protocol is determined so that the packet is ready for transport ("layer 4") processing. For each transport layer protocol, a corresponding entry handler function is defined: *tcp_v4_rcv()* and *udp_rcv()* are two examples. When a packet is passed upwards, the corresponding protocol entry handler function is called.

2.2.2 TCP Processing

When a packet (TCP segment) is handed upwards for TCP processing, the function *tcp_v4_rcv()* first performs the TCP header processing. Then *__tcp_v4_lookup()* is called to find the corresponding *socket* that is associated with the packet. A packet without a corresponding *socket* will be dropped. A socket has a lock structure to protect it from unsynchronized 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* [27]. 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. Except in the case of prequeue overflow, those queues 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 or 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 the *tcp_established* state, *tcp_rcv_established*() is called; otherwise, *tcp_rcv_state_process()* is called to handle state transitions and connection management, if there are no header or checksum errors. *tcp_rcv_established()* performs key TCP actions such as sequence number checking, RTT estimation, acknowledging, and data packet processing. Here, we focus on the data packet processing.

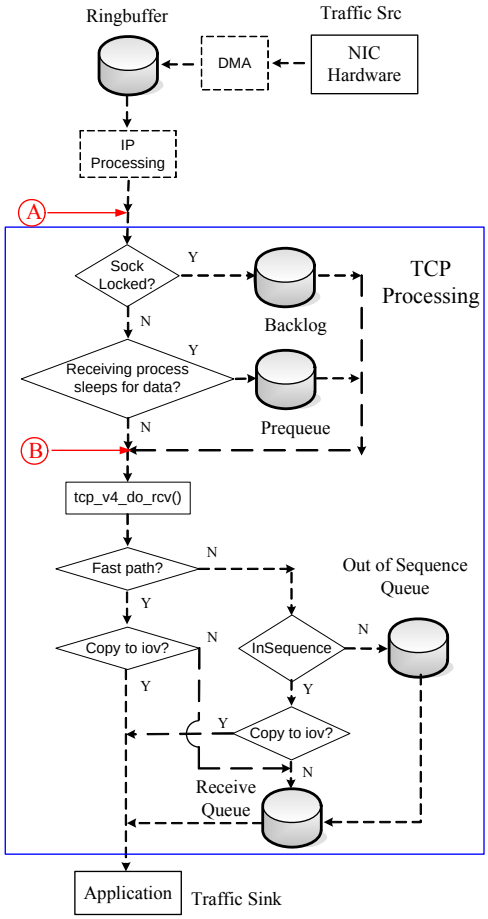

Figure 3 TCP Processing - Interrupt Context

When a data packet is handled on the *fast path*, *tcp_rcv_established()* checks 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 a system call such as *recvmsg()*. The conditions for checking whether to deliver the data packet to the user space are as follow:

- The socket belongs to the currently active process; AND
- The current packet is the next in sequence for the socket; AND
- 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 (packet is the next one to be delivered to the user). 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 acknowledged, 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*. In the figure, "A" and "B" stand for measurement points that will be referred to in later sections.

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 eventually lead to *tcp_recvmsg(),* which is the top end of the TCP transport receive mechanism. As shown in Figure 4, *tcp_recvmsg()* first locks the socket, then checks the *receive queue*. Since packets in the receive queue are guaranteed in order, acknowledged, 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()* does not return to user space until the *prequeue* and *backlog queue* are drained. *tcp_recvmsg()* may need to fetch a certain amount of data before it return 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.

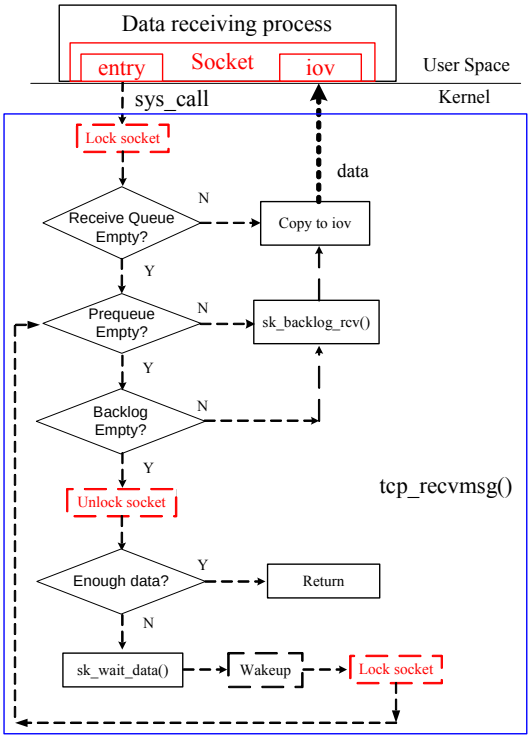

Figure 4 TCP Processing – Process Context

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 in section 2.2.2, all the TCP socket-related receive system calls result in a call to *tcp_recvmsg(),* which will copy packet data from socket's buffers (*receive queue*, *prequeue*, *backlog queue*) through *iovec.*

3. Potential Performance Bottleneck for TCP

As described above, TCP processing can be performed in interrupt or process context, depending on the status of the TCP receive socket and the data receiving process. To summarize, the different TCP packet processing scenarios are as shown in Figure 5. Since Linux is an interrupt-driven operating system, interrupt processing has higher priority than user-lever processes. TCP packets handled in the interrupt context are usually

processed immediately by TCP protocol engine, independent of system load. However, when incoming TCP packets are on the prequeue or backlog queue, those packets will be handled in the process context. In that case, TCP processing is strongly related to system load and the Linux process-scheduling scheme. This leads to a potential performance bottleneck for TCP applications when the system is under load.

Process \ Socket	locked	unlocked
sleep (by *sk_wait_data()*)	N/A	Wait on prequeue, (process context)
run	Wait on backlog, (process context)	Process immediately (interrupt context)

Figure 5 TCP Packets Processing Scenarios

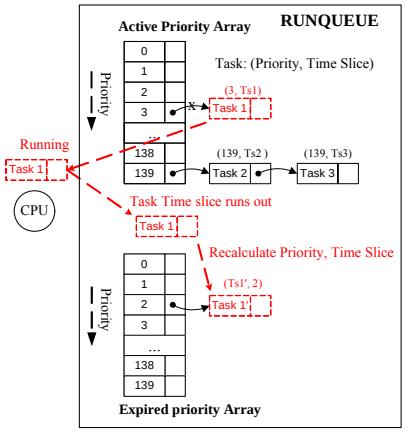

Figure 6 Linux Process Scheduling

Linux 2.6 is a *preemptive multi-processing* operating system. Processes (tasks) are scheduled to run in a *prioritized round robin* manner [27][29][30], to achieve the objectives of fairness, interactivity and efficiency. For the sake of scheduling, a Linux process has a dynamic priority and a static priority. A process' static priority is equivalent to its *nice* value, which is specified by the user in the range –20 to +19 with a default of zero, and not changed by the kernel [27][29]. Higher values correspond to lower priorities. The dynamic priority is used by the scheduler to rate the process with respect to the other processes in the system. An eligible process with a better (smaller-valued) dynamic priority is scheduled to run before a process with a worse (higher-valued) dynamic priority. The dynamic priority varies during a process' life. It depends on a dynamic priority bonus, from -5 to +5, and its static priority. The dynamic priority bonus depends on the process' interactivity status. Linux credits interactive processes and penalizes non-interactive processes by adjusting this bonus. There are 140 possible priority levels in Linux. The top 100 levels (0-99) are used only for real-time processes, which we do not address in this paper. The last 40 levels (100-139) are used for conventional processes.

As shown in Figure 6, process scheduling employs a data structure called *runqueue*. Essentially, a runqueue keeps track of all runnable tasks assigned to a particular CPU. 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 a queue of runnable processes per priority level. Higher (dynamic) 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. Table 1 shows the time slices for various nice values. When a process in the active priority array runs out of its time slice, it is considered *expired* and moved to the expired priority array if it is not interactive, or reinserted back into the active array if it is interactive. During the move, a new time slice and dynamic priority are calculated. When there are no more runnable tasks in the active priority array,

it is simply swapped with the expired priority array. A running process might be put into a wait queue to sleep, waiting for expected events (e.g., I/O). When a sleeping process wakes up, its time slice and priority are recalculated and it is moved to the active priority array. As for preemption, whenever a scheduler clock tick or interrupt occurs, if a higher-priority task has become runnable, it will preempt the running task if the latter holds no kernel locks.

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

Table 1 Nice value vs. Time slice

Furthermore, when a process is termed interactive, its time slice is divided into smaller pieces. When it finishes a piece, the task will round robin with other tasks at the same priority level. This way execution will rotate more frequently among interactive tasks of the same priority, preventing interactive processes from blocking other interactive processes of the same priority.

Under the simplifying assumption that processes other than the data receiving process of interest do not sleep for long times, and hence use their entire time slices before the active priority array is empty, let's first consider the backlog scenario. The data receiving process is calling *tcp_recvmsg()* to fetch packet data from socket receive buffer to user space. The socket will be locked until the process returns to the user space. If the data receiving process' time slice ends before the lock is released, the data receiving process will be moved to the expired priority array with the socket locked. The socket will remain locked until the lock is released after the data receiving process resumes its execution in the next round of running. The time until the process resumes its execution is strongly dependent on the system load. Let's assume that when the expired data receiving process is moved to the expired priority array, there are, in all, N_1 running processes (P_1, ..., P_{N_1}) left in the active array, and N_2 expired processes (P_{N_1+1}, ..., $P_{N_1+N_2}$) in the expired array with priorities higher than that of the expired data receiving process. Considering that some of the N_1 processes, when expired, might move to the expired array with recalculated priorities higher than that of the expired data receiving process, the minimum time before the data receiving process could resume its execution would be:

$$\sum_{j=1}^{N_1+N_2} Timeslice(P_j) \qquad (1)$$

Here, $Timeslice(P_j)$ denotes the time slice of process P_j.

As we have seen, during this period all the new incoming TCP packets for the data receiving process will wait on the socket's backlog queue without being TCP processed. No acknowledgements will be fed back to the TCP sender.

As for the prequeue scenario, the data receiving process might sleep within *tcp_recvmsg()* waiting for data. Before the process wakes up, all the incoming segments for the data receiving process will wait on the prequeue without being TCP-processed. Let's assume that when the woken-up data receiving process is moved to the active prior-

ity array, there are N_3 other processes in the active array whose priorities are higher. Assuming still that each process will use its full time slice, the minimum time before the data receiving process could resume its execution would be:

$$\sum_{j=1}^{N_3} Timeslice(P_j) \tag{2}$$

Similarly, during this period no acknowledgements will be returned to the TCP sender.

Note that the data receiving process might be preempted within *tcp_recvmsg()* by other higher priority processes. In this case, packet might wait on the backlog queue. The analysis of how long packets would wait on the backlog queue is similar to the above.

Let's denote by T_{wait} the time a packet waits in the backlog queue or prequeue. In the worst case, we have

$$T_{wait} > \sum_{j=1}^{N_1+N_2} Timeslice(P_j) \qquad \textit{for backlog queue} \tag{3-1}$$

$$T_{wait} > \sum_{j=1}^{N_3} Timeslice(P_j) \qquad \textit{for prequeue} \tag{3-2}$$

The time that packets wait on the backlog queue or prequeue adds to the sender's estimate of the round-trip time (RTT), since ACKs have not been sent for segments on those queues.

Usually it is the case that:

$$RTT = T_t + T_{pd} + T_q + T_{pp} \tag{4}$$

Where T_t is the time the interface takes to send the packet. This will likely have a linear dependence on packet size. T_{pd} is the time taken for a signal to propagate through the network medium. In a single simple network link, this is equal to the distance between sender and receiver divided by propagation speed. T_q is the time spent in routing queues along the path. This will fluctuate over time depending on congestion in the path. And T_{pp} is the amount of time spent by sender and receiver and routers doing processing necessary for packet delivery. On current hardware, this is usually in the sub-microsecond range. For a given packet size, network path, sender, and receiver, it can be assumed that T_t, T_{pd}, and T_{pp} are constants. For packet switched data networks, T_q is usually a random variable, following some distribution. Hence, RTT is treated as a random variable.

Since TCP packets might wait on the backlog queue or prequeue in the receiver, we will have:

$$RTT = T_t + T_{pd} + T_q + T_{pp} + T_{wait} \tag{5}$$

Clearly, if TCP packets are processed in interrupt context, $T_{wait} \approx 0$. In the receiver, since the system load is uncertain, whether, when, and how long TCP packets might wait on the backlog queue or prequeue is uncertain, and T_{wait} is also a random variable, following some distribution. We can see that T_{wait} and T_q are independent, or effectively so.

Hence, it will be the case that:

$$E(T_t + T_{pd} + T_q + T_{pp} + T_{wait}) > E(T_t + T_{pd} + T_q + T_{pp}) \tag{6}$$

$$Var(T_t + T_{pd} + T_q + T_{pp} + T_{wait}) > Var(T_t + T_{pd} + T_q + T_{pp}) \tag{7}$$

Here, $E(*)$ and $Var(*)$ are the expected value and variance of a random variable, respectively. From (6) and (7), it can be derived that when packets wait on backlog queue or prequeue, both RTT and its variance will increase.

In [31], Mathis et al., derive:

$$BW = \frac{MSS}{RTT}\frac{C}{\sqrt{p}} \tag{8}$$

Where BW is the achievable bandwidth from sender to receiver, MSS is the maximum segment size, C is a constant of order unity, and p is the packet drop probability along the path. Note: (8) is based on Reno TCP.

It follows from (6) and (8) that with increased RTT, the average achievable bandwidth from sender to receiver will decrease. Also, as we know, when competing TCP network traffic exists, increased RTT will put a TCP data stream in a disadvantaged position [32].

The TCP sender does not measure RTT precisely, but rather maintains "smoothed" estimates SRTT and *RTTVAR* of round-trip time and its variation, and uses the estimates in determining the *RTO*, or retransmit timeout period [33][34] after which an unacknowledged packet is assumed lost and will be retransmitted. Estimates are updated as follows whenever a new measurement *R* of round-trip time is available from the acknowledgement of a timed data segment:

$$RTTVAR := \frac{3}{4}RTTVAR + \frac{1}{4}|SRRT - R| \tag{9}$$

$$SRTT := \frac{7}{8}SRTT + \frac{1}{8}R \tag{10}$$

$$RTO := \max\{TCP_RTO_MIN, \min\{SRTT + 4 \times RTTVAR, TCP_RTO_MAX\}\} \tag{11}$$

The variation defined by (9) is not variance in the strict statistical sense, but is more easily calculated in the kernel and is commonly referred to as variance. Here, both TCP_RTO_MIN and TCP_RTO_MAX are constants, which are 200ms and 120s respectively. Experience has shown that clock granularity affects RTO calculation. Finer granularity (≤100ms) performs somewhat better than coarser granularities [34]. In Linux 2.6, the clock granularity is not a big concern, since it has reached the 1ms level.

Also, from (6), (7), and (11), it can be seen that when the RTT's variance rises, the RTO in the sender will correspondingly increase. In that case, the TCP sender may be less responsive to packet losses, resulting in degraded performance.

When packets wait on the receiver's backlog queue or prequeue too long, it triggers the retransmission timeout in the sender. Assuming that when packets start to wait on the queue, the current RTO value in the sender is T_{RTO}. The sender will time out when:

$$T_{wait} > T_{RTO} - T_t - T_{pd} - T_q - T_{pp} \quad (12)$$

If the system load is medium or high, condition (12) can be easily met. For example, if $N_1 + N_2 \geq 10$, and all the running processes have the default nice value of 0, from equations (1) and (3-1) and Table 1, we can easily have $T_{wait} > 1s$, large enough to outrigger the retransmission timer. Once RTO times out, the sender incorrectly assumes packet loss in the network. Such spurious timeouts affect TCP performance in two ways: (1) the sender unnecessarily reduces its offered load by setting its congestion window size to 1 segment; (2) the sender is forced into a go-back-N retransmission model. Spurious timeouts are usually due to sudden delay spike in the path, e.g. route changes. The Eifel algorithm [35] and F-RTO algorithm [36] have been proposed to solve the spurious timeout problem in the sender. However, since the spurious timeout in the case at hand is caused by Linux TCP implementation in the receiver, we seek a solution in the receiver.

4. Experiments and Analysis

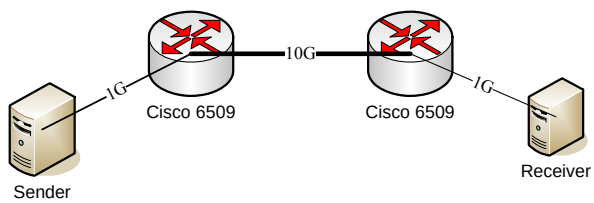

Figure 7 Experiment Network & Topology

To verify our claims in section 3, we ran data transmission experiments upon Fermilab's sub-networks. In the experiments, we run *iperf* [37] to send data in one direction between two computer systems. *iperf* in the receiver is the data receiving process. As shown in Figure 7, the sender and the receiver are connected to two Cisco 6509 switches connected to each other by an uncongested 10-gigabit/second link. During the experiments, the background traffic in the network is low, and there is no packet loss, or packet reordering in the network. The sender and receiver's features are as shown in table 2.

	Sender	Receiver[+]
CPU	Two Intel Xeon CPUs (3.0 GHz)	One Intel Pentium III CPU (1 GHz)
System Memory	3829 MB	512MB
NIC	Tigon, 64bit-PCI bus slot at 66MHz, 1Gbps/sec, twisted pair	Syskonnect, 32bit-PCI bus slot at 33MHz, 1Gbps/sec, twisted pair

Table 2 Sender and Receiver Features

[+] We ran experiments on different types of Linux receivers in Fermilab, and similar results were obtained.

In order to study the detailed packet receiving process, we have added instrumentation within Linux packet receiving path. Specifically, to collect statistics and provide insights for T_{wait}, we have added measurement points A and B in Linux packet receiving path as shown in Figure 3. For each packet, the times that it passes over point A (t^A) and point B (t^B) are recorded; we collect the statistics for the time difference $t^{diff} = t^B - t^A$. It can be assumed that $T_{wait} \approx t^{diff}$, to within a few CPU cycles. Since it is difficult to keep track of every packet, we classify t^{diff} into six different groups: $0 \le t^{diff} \le 1ms$, $1ms \le t^{diff} \le 10ms$, $10ms \le t^{diff} \le 0.1s$, $0.1s \le t^{diff} \le 0.2s$, $0.2s \le t^{diff} \le 1s$, and $t^{diff} > 1s$. We collect the histogram for each category.

To create a variable system load in the receiver, we compiled the Linux Kernel with *n* parallel tasks by running `make -nj` [27]. The different value of *n* corresponds to different level of background system load, e.g. `make -10j`. For simplicity, they are termed as "BL*n*". The background system load implies load on both CPU and system memory. In the TCP experiments, sender transmits one TCP stream to receiver for 20 seconds. All the processes are running with a nice value of 0, and iperf's receive buffer is set to 40MB. In the sender, we use *tcpdump* to record tcp streams, and later use *tcptrace* [38] to analyze the recorded packets. Consistent results were obtained across repeated runs. In the following sections, we present two groups of experiments, with background loads of BL0 and BL10 respectively. The experimental data are from both sender and receiver side. Table 3 shows the iperf output results in the sender.

System Load in Receiver	TIME	Data transmitted	End-to-End Throughput
BL0	20 sec	1.17Gbytes	504Mbits/s
BL10	20 sec	174Mbytes	72.1Mbits/s

Table 3 iperf output results

Figure 8 shows the time-sequence diagrams for the recorded TCP traces from the sender side. Figure 8a shows that with a background load of BL0, the sender sends packets smoothly and continuously. Packets are acknowledged in time and no RTO happens. However, the time sequence diagram in Figure 8b shows another story. With BL10 in receiver, sender sends packets intermittently. In the diagram, the small red "R" represents retransmission. There are quite a few retransmissions. Since there are no packet losses or reordering in the network, those unnecessary retransmissions are due to spurious timeouts in the sender. As we have analyzed in section 3, when packets wait on backlog queue and prequeue too long, no ACKs are returned to the TCP sender in time, leading to RTOs in the sender.

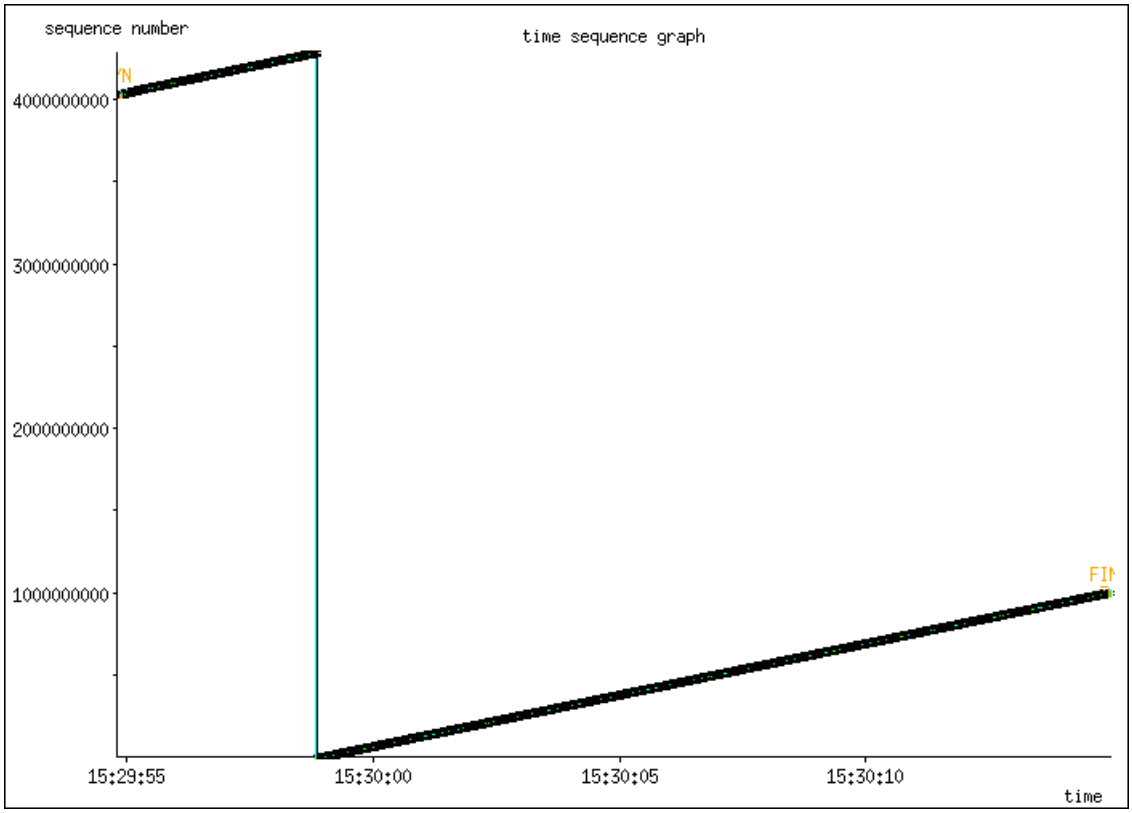

Figure 8a Sender Time Sequence Diagram (Sender Side), with BL0 in receiver

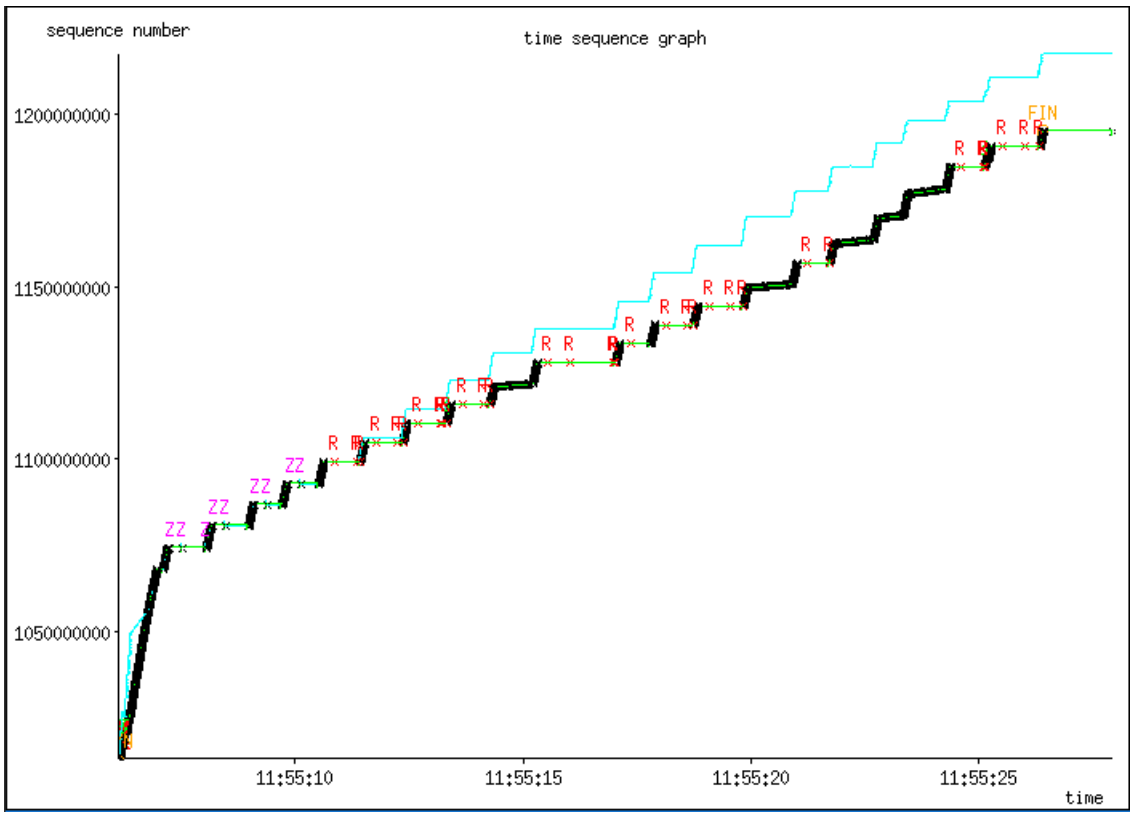

Figure 8b Sender Time Sequence Diagram (Sender Side), with BL10 in receiver

Now let's study the issues from the receiver side, to further verify the conclusions of section 3.

Figures 9 and 10 show various TCP queues at BL0 and BL10 respectively.

- Normally, prequeue and out-of-sequence queue are empty. The backlog queue is usually not empty, even during the periods that the data receiving process is not running. Packets are not dropped or reordered in the test network. However, packets might be dropped by the NIC in the reception ring buffer [39], causing subsequent packets to go to the out-of-sequence queue.
- With BL0, the receive queue is approaching full. In our experiment, since the sender is more powerful than the receiver, this scenario is as expected. The experiment results have confirmed this point. With BL10, since RTOs in the sender seriously degrading the TCP performance, the receive queue is not approaching full, even with a background load of BL10
- In contrast with Figure 9, the backlog and receive queues in Figure 10 show some kind of periodicity. The periodicity matches the data receiving process' running cycle [39]. In Figure 9, with BL0, the data receiving process runs almost continuously, but at BL10, it runs only intermittently.

Though Figures 9 and 10 provide information about status of various TCP queues, they provide no clue about how long packet will wait on backlog queue or prequeue. Table 4 gives the statistics about t^{diff} with BL0 and BL10. As we have known that there is $T_{wait} \approx t^{diff}$, the statistics about t^{diff} will provide us further information about T_{wait}. When the load is light in the receiver, packets would go to prequeue or backlog queue (as shown in backlog queue of Figure 9). Since the data receiving process run continuously, packets within backlog queue or prequeue are processed soon, T_{wait} won't be large. However, when the system load increases, as it has been analyzed in formula (3-1) and (3-2), T_{wait} might be large, which has been confirmed by Table 4. As shown in Table 4, with BL0, there are no packets with $t^{diff} \geq 10ms$. However, with BL10, some packets even have $t^{diff} \geq 1s$, waiting on the backlog queue or prequeue for quite a long period of time, without being TCP-processed. This is why we have seen in Figure 8b that RTO occurs.

System Load	< 1ms	1ms–10ms	10ms–100ms	100ms–200ms	200ms-1s	>1s
BL0	862429	636	0	0	0	0
BL10	122657	744	896	40	300	75

Table 4 t^{diff} statistics (Receiver Side)

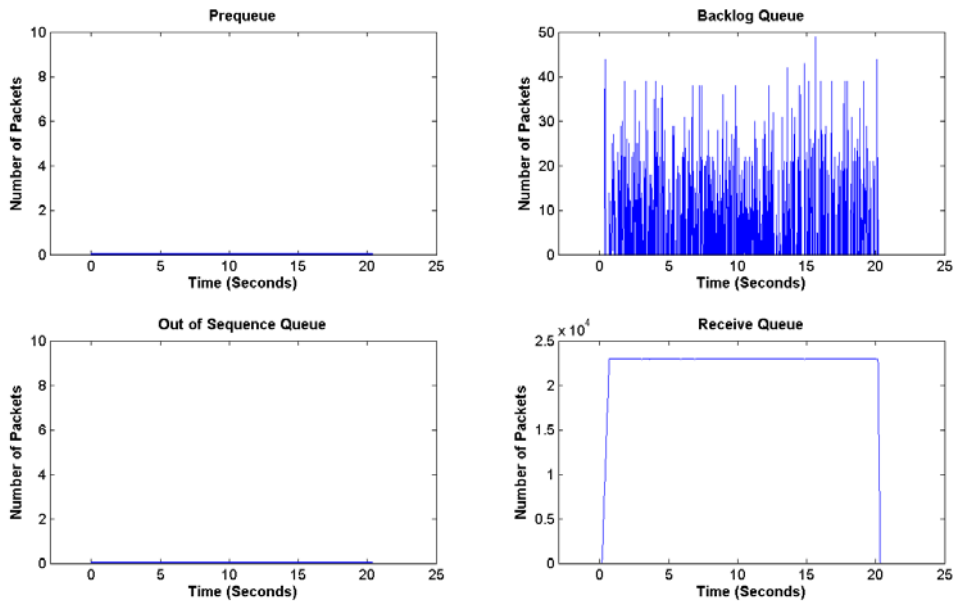

Figure 9 Various TCP Receive Buffer Queues – BL0 (Receiver Side)

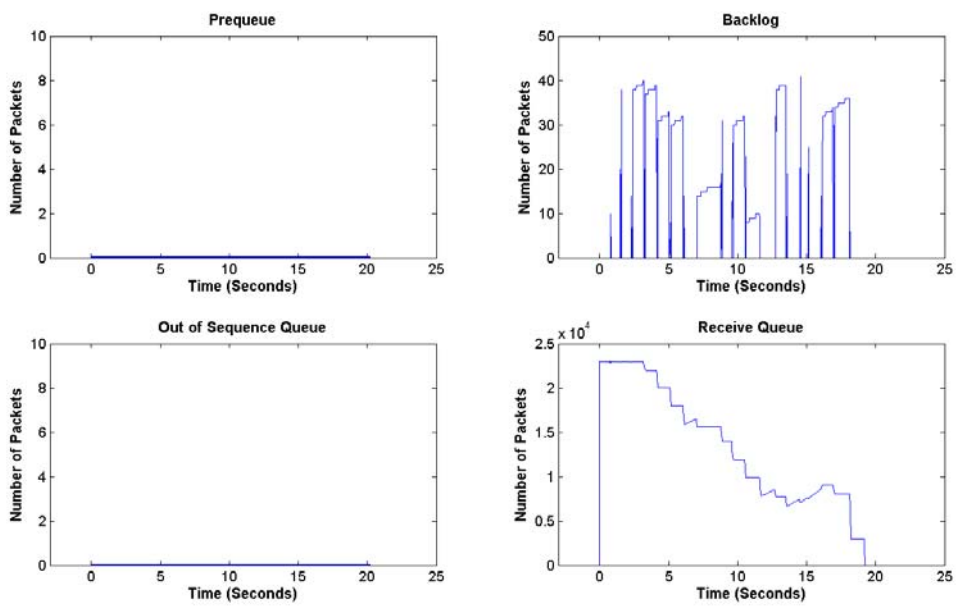

Figure 10 Various TCP Receive Buffer Queues – BL10 (Receiver Side)

5. A Possible Solution

As described above, the TCP performance bottleneck is due to the fact that TCP packets might wait on the backlog queue or prequeue in the receiver without being TCP-processed. To resolve the performance bottleneck issue, there might be two basic approaches. Naturally, the first approach is to always do TCP processing in the interrupt context, not in the process context at all. However, this would require the overhaul of the whole Linux TCP protocol engine, which might be complex and time-consuming. The second approach is to reduce T_{wait} for packets waiting on prequeue or backlog queue. As

implied by formulas (1), (2), and (3), the underlying idea here is that when there are packets waiting on the prequeue or backlog queue, do not allow the data receiving process to release the CPU for long. Relatively, the second approach is easier to implement. We have modified the Linux process scheduling policy and *tcp_recvmsg()* to implement a solution of the second sort. The pseudo code for scheduling is as shown in Listing 1. The code changes for *tcp_recvmsg()* is as shown in Listing 2, highlighted in red. To summarize, the solution works as follows: an expired data receiving process with packets waiting on backlog queue or prequeue is moved to the active array, instead of expired array as usual. More often than not, the expired data receiving process will continue to run. Even it doesn't, the wait time before it resumes its execution will be greatly reduced. However, this gives the process extra runs compared to other processes in the runqueue. For the sake of fairness, the process would be labeled with the *extra_run_flag*. Also considering the facts that: (1) the resumed process will continue its execution within *tcp_recvmsg()*; (2) *tcp_recvmsg()* does not return to user space until the *prequeue* and *backlog queue* are drained. For the sake of fairness, we modified *tcp_recvmsg()* as such: after *prequeue* and *backlog queue* are drained and before *tcp_recvmsg()* returns to user space, any process labeled with the *extra_run_flag* will call *yield()* to explicitly yield the CPU to other processes in the runqueue. *yield()* works by removing the process from the active array (where it currently is, because it is running), and inserting it into the expired array [27]. Also, to prevent processes in the expired array from starving, a special rule has been provided for Linux process scheduling (the same rule used for interactive processes [29]): an expired process is moved to the expired array without respect to its status if processes in the expired array are starved.

```
If (process->timeslice - - == 0) {
        recalculate timeslice and priority;
        if (packets are waiting on backlog queue or prequeue) {
                if (processes in expired array are starved)
                        move the process to expired array;
                else {
                        move process to active array;
                        if (process is non-interactive)
                        set process->extra_run_flag:=TRUE;
                }
        }
        else {
                … as usual …
        }
}
else {
        … as usual …
}
```

Listing 1 Pseudo code for scheduling policy

```
tcp_recvmsg{

        … as usual …

        TCP_CHECK_TIMER(sk);
        release_sock(sk);

        if (process->extra_run_flag == TRUE){
                set process->extra_run_flag:=FALSE;
                yield();
        }

        return copied;

        … as usual …
}
```

Listing 2 Code changes for *tcp_recvmsg()*

We repeated the TCP experiments as described in section 4 on Linux updated with the new scheduling policy described as above. We compare the new experiment data with those obtained in section 4. The old experiment data will be prefixed with “O-”; whereas, the new data is prefixed with “N-”.

Table 5 shows the iperf output results in the sender. It can be seen that the TCP performance of N-BL10 is much better than that of O-BL10. The TCP end-to-end throughput of N-BL10 reaches as high as 88.8Mbits/s; however, with O-BL10, the corresponding TCP end-to-end throughput is only 72.1Mbits/s. This implies that our proposed solution is effective in resolving the TCP performance bottleneck issue. This point is verified by experiment data from both sender and receiver sides. Figure 11 shows the time-sequence diagrams for the recorded TCP traces from the sender side. For comparison, Figure 11a shows old kernel’s time sequence diagram with a background load of BL10. And Figure 11b shows the time sequence diagram with N-BL10. In Figure 11b, there are no retransmissions due to packets waiting on backlog queue or prequeue too long; packets are acknowledged in time and no RTO happens. Nevertheless, the sender still transmits intermittently. This is caused by zero window advertisements from receiver. In our experiments, sender is a more powerful machine than the receiver; in addition, the receiver runs with a high background load. When packets cannot be consumed by the data receiving process in time, the data receive buffer in receiver is approaching full. Then receiver will advertise zero windows to stop sender transmitting. The small purple “Z” in Figure 11b represents a window advertisement of 0 bytes received from the receiver. Later, from Figure 12 we can see that the receive buffer is approaching full.

System Load in Receiver	TIME	Data transmitted	End-to-End Throughput
O-BL0	20 sec	1.17Gbytes	504Mbits/s
O-BL10	20 sec	174Mbytes	72.1Mbits/s
N-BL10	20sec	220Mbytes	88.8Mbits/s

Table 5 iperf output results

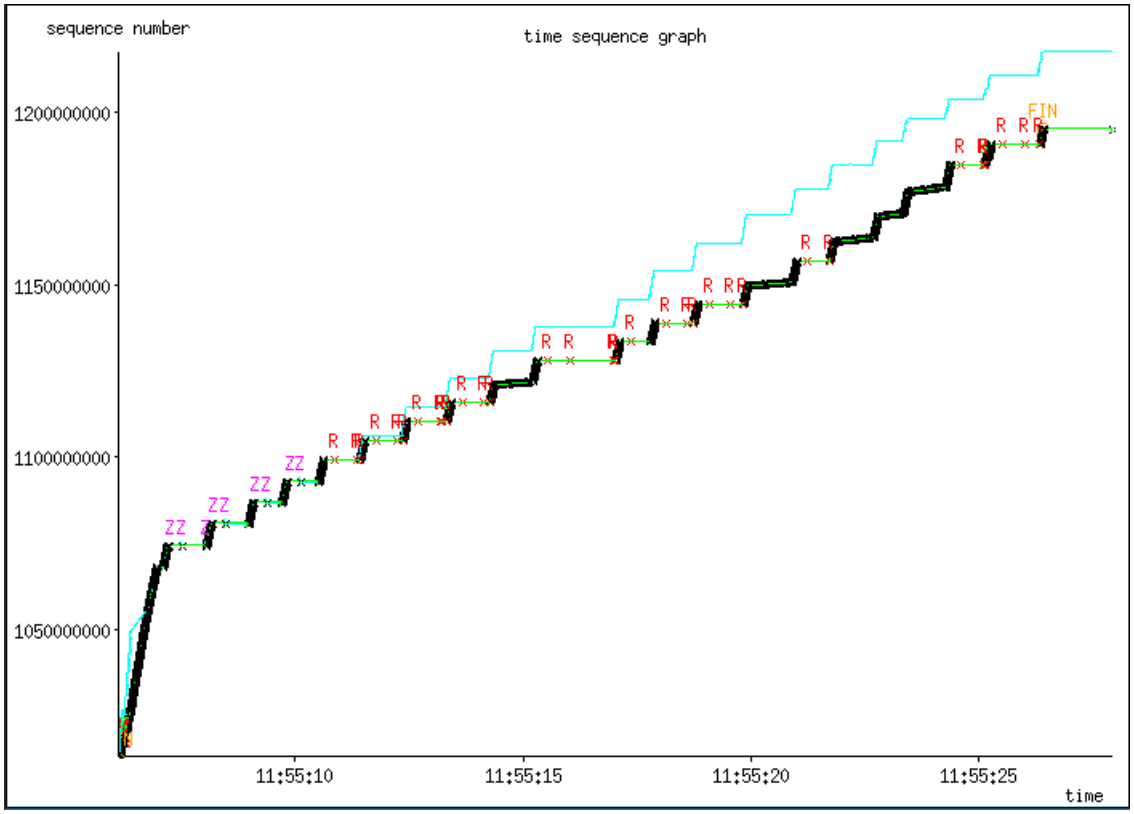

Figure 11a Sender Time Sequence Diagram (Sender Side), with BL10 in receiver (OLD kernel)

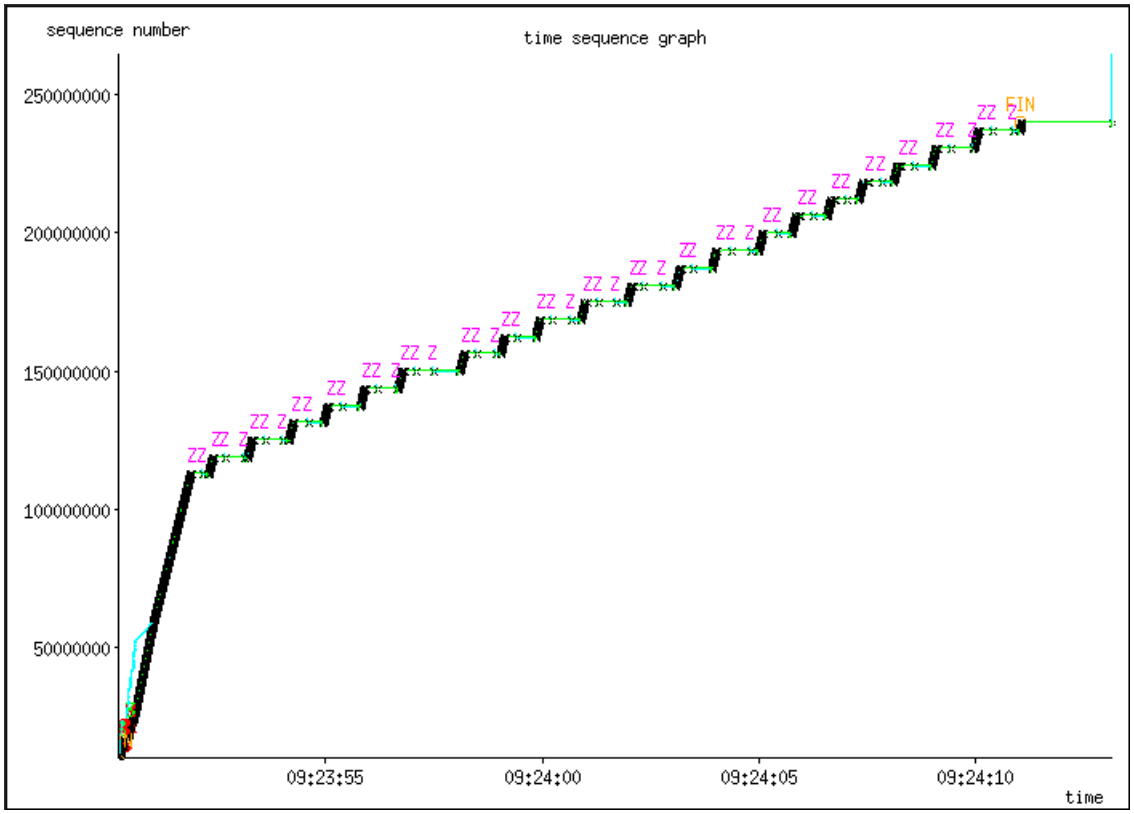

Figure 11b Sender Time Sequence Diagram (Sender Side), with BL10 in receiver (NEW kernel)

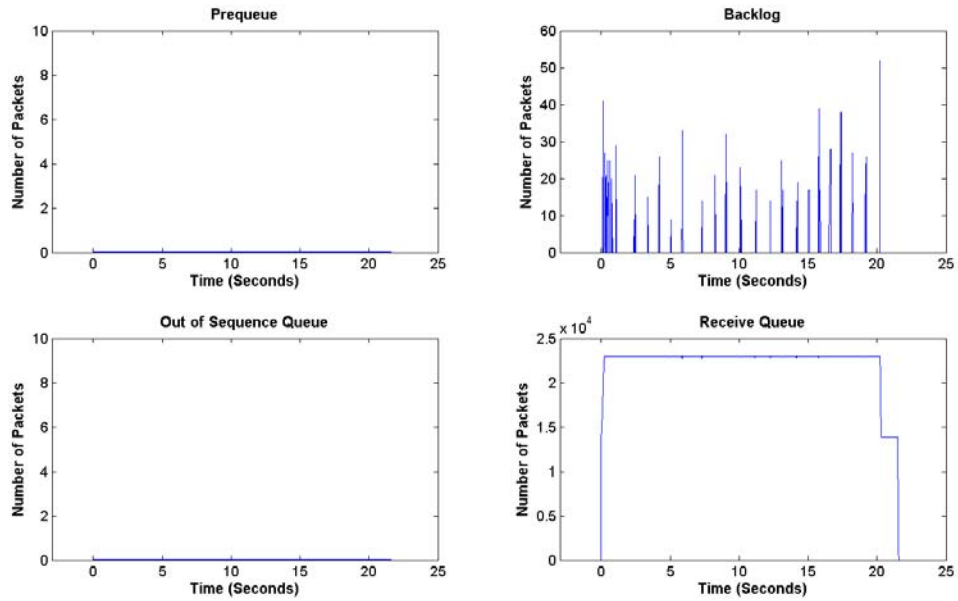

Figure 12 Various TCP Receive Buffer Queues – BL10 (NEW kernel, Receiver)

Figures 12 shows various TCP queues with N-BL10. It can be seen that the receive queue is approaching full. Compared with Figure 10, we can be concluded that TCP performance is really enhanced. Table 6 gives observations of t^{diff} for N-BL10. There are now no packets with $t^{diff} \geq 200ms$, which implies that no packets waited long on the prequeue or backlog queue. It further verifies that our proposed solution is effective in resolving the TCP performance bottleneck issue.

System Load	< 1ms	1ms–10ms	10ms–100ms	100ms–200ms	200ms-1s	>1s
O-BL0	862429	636	0	0	0	0
O-BL10	122657	744	896	40	300	75
N-BL10	156300	851	618	29	0	0

Table 6 t^{diff} statistics (Receiver Side)

Now, let's study the fairness issues of our proposed solution. Readers might suspect that our proposed solution might cause fairness issues. The following experiments and analysis will show that it does not significantly do so.

As described above, Linux scheduler moves an expired process to the expired priority array if the process is not interactive, or reinserts it back into the active array if the process is interactive. To better evaluate the fairness performance of our proposed solution, we try to eliminate the influence of interactive processes in the following experiments: non-interactive processes run as background loads. To create non-interactive processes in the receiver, we develop a CPU intensive application that executes a number of operations in a loop. In all the following experiments, the sender transmits one TCP stream to the receiver for 20 seconds. In the receiver, iperf is run as "***time iperf –s –w 20M***". All the processes are running with a nice value of 0. Further, since the transmission lasts 20 seconds, in the receiver we calculate iperf's experiment CPU share as:

$(stime + utime)/20s$. We compare the iperf's experiment CPU shares with its fair CPU share. If there are *M* background processes, iperf's fair CPU share is: *1/(M+1)*. Consistent results were obtained across repeated runs. In the following sections, we present a group of experiment results in Table 7.

Background Processes in Receiver	Iperf Experiment Results				
	End-to-End Throughput	Utime	Stime	Iperf Experiment CPU Share	Iperf Fair CPU Share
1 processes	265Mbps	0.048s	10.47s	52.58%	50%
3 processes	143Mbps	0.028s	5.644s	28.38%	25%
4 processes	117Mbps	0.032s	4.728s	23.8%	20%
9 processes	70Mbps	0.028s	2.744s	13.86%	10%

Table 7 Fairness Experiments

As shown in Table 7, in the worst case, iperf gains the extra CPU shares of 3.86%. Considering the possibilities that iperf itself might sometimes be termed interactive and gain extra runs, the experiment results show that our proposed solution will not cause fairness issues. The proposed solution tradeoffs a small amount of fairness performance to resolve the TCP performance bottleneck. The reason that our proposed solution will not cause serious fairness issues is due to the facts that:

(1) Each time when an expired data receiving process with packets waiting on backlog queue or prequeue is moved to the active array, it gains at most the *tcp_recvmsg()* amount of extra time compared to other processes in the runqueue;
(2) Each calling of *tcp_recvmsg()* will not take long. When Linux kernel processes packets within backlog queue or prequeue, the processed data will be fed to the socket out of sequence queue or receive queue, then TCP flow control will take effect to slow down or throttle sender.
(3) The possibility that a data receiving process runs out of its timeslice and is moved to the expired array with packets waiting on backlog queue or prequeue does not occur often, compared to the Linux scheduling time scale. This has been shown in Figure 8b. As iperf does pure data transmission, the received data will not be further processed in the user space. Therefore, for a real network application, this possibility is even lower.

Another justification for our proposed solution is that "Fairness is often a hard attribute to justify maintaining because it is often a tradeoff between over global performance and localized performance. For example, in an effort to provide maximum disk throughput, the Linux 2.4 block I/O scheduler may starve older requests, in order to continue processing newer requests at the current disk head position. This minimizes seeks and thus provides maximum overall disk throughput – at the expense of fairness to all requests" [40].

6. Conclusions

Suspension of TCP processing for incoming packets induces both an increase and a greater variability of the round-trip time measured by the sender. A moderate or high sys-

tem load on the receiver can delay TCP processing so long as to cause a timeout in the sender, which will then resume sending at the minimum possible rate. Current storage implementations in the High Energy Physics community and elsewhere exploit the disk space of compute-farm worker nodes [41], making this a very topical concern.

So far, we have been discussing how the proposed solution behaves in improving TCP performance, and resolving the bottleneck. Also, we evaluate the fairness performance of our proposed solution. The proposed solution trades a small amount of fairness performance to resolve the TCP performance bottleneck. Our experiments and analysis have shown that our proposed solution won't cause serious fairness issues. The criteria for a good scheduling algorithm also include efficiency, response time, turnaround, and throughput [42]. How our first cut at a throughput-enhancing process scheduling policy affects other processes and overall system performance needs further study. We will cover this topic in other papers.

References

[1] W. Allcock *et al.*, "The Globus Striped GridFTP Framework and Server," Proceedings of the ACM/IEEE Supercomputing 2005 Conference, Seattle, USA, 2005.

[2] I. Foster *et al.*, The Grid: Blueprint for a New Computing Infrastructure, Second Edition, New York: Wiley, 2004.

[3] F. Berman *et al.*, Grid Computing: Making the Global Infrastructure a Reality, New York: Wiley, 2003.

[4] V. Jacobson, "Congestion Avoidance and Control," in Proc. ACM SIGCOMM, Stanford, CA, Aug. 1988, pp. 314 – 329.

[5] K. Fall *et al.*, "Simulation-based Comparison of Tahoe, Reno and SACK TCP," ACM Computer Communications Review, vol. 5, no. 3, pp. 5--21, 1996.

[6] R. Braden *et al.*, "TCP Extensions for Long Delay Paths," RFC 1072, 1988.

[7] R. Braden *et al.*, "TCP extensions for high-speed paths," RFC 1185, Oct. 1990

[8] L. S. Brakmo *et al.*, "TCP Vegas: End to End Congestion Avoidance on a Global Internet", IEEE Journal on Selected Areas in Communications, vol. 13, no. 8, pp. 1465 – 1480, 1995.

[9] C. Casetti *et al.*, "TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links", Proceedings of ACM Mobicom 2001, pp. 287-297, Rome, Italy, 2001.

[10] M. Gerla *et al.*, "TCP Westwood: Congestion Window Control Using Bandwidth Estimation," Proceedings of IEEE Globecom 2001, vol. 3, pp. 1698-1702, San Antonio, USA, 2001.

[11] C. Jin *et al.*, "FAST TCP: From Theory to Experiments," IEEE Network, vol. 19, no. 1, pp. 4-11, January/February 2005.

[12] L. Xu *et al.*, "Binary Increase Congestion Control for Fast Long-distance Networks," Proceedings of IEEE INFOCOM 2004, Hong Kong, 2004.

[13] D. J. Leith, *et al.*, "H-TCP: A Framework for Congestion Control in High-speed and Long-distance Networks," Hamilton Institute Technical Report, August 2005.

[14] S. Floyd, "HighSpeed TCP for Large Congestion Windows," RFC 3649, December 2003.

[15] M. K. Gardner *et al.*, "User-space Auto-tuning for TCP Flow Control in Computational Grids," Computer Communications, vol. 27, no. 14, pp. 1364-1374, 2004.

[16] J. Widmer *et al.*, "A Survey on TCP-Friendly Congestion Control," IEEE Network Magazine, Special issue on Control of Best Effort Traffic, vol. 15, no. 3, pp. 28-37, 2001.

[17] J. Martin *et al.*, "Delay-Based Congestion Avoidance for TCP," IEEE/ACM Transactions on Networking, vol. 11, no. 3, June 2003.

[18] M. Mathis *et al.*, "Forward Acknowledgment: Refining TCP Congestion Control," Proceedings of SIGCOMM'96, August 1996, Stanford, CA.

[19] 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.

[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] M. Mathis *et al.*, "Web100: Extended TCP Instrumentation for Research, Education and Diagnosis," ACM Computer Communications Review, vol. 33, no. 3, July 2003.
[22] T. Dunigan *et al.*, "A TCP Tuning Daemon," Proceedings of the ACM/IEEE Supercomputing 2002 Conference, Baltimore, USA, 2002.
[23] M. Rio *et al*., "A Map of the Networking Code in Linux Kernel 2.4.20," March 2004.
[24] 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.
[25] K. Wehrle *et al.*, The Linux Networking Archetecture – Design and Implementation of Network Protocols in the Linux Kernel, Prentice-Hall, ISBN 0-13-177720-3, 2005.
[26] www.kernel.org.
[27] R. Love, Linux Kernel Development, Second Edition, Novell Press, ISBN: 0672327201, 2005.
[28] J. Corbet *et al.*, Linux Device Drivers, Third Edition, O'Reilly Press, ISBN: 0-596-00590-3, 2005.
[29] D. P. Bovet *et al.*, Understanding the Linux Kernel, Third Edition, O'Reilly Press, ISBN: 0-596-00565-2, 2005.
[30] 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.
[31] M. Mathis *et al.*, "The Macroscopic Behavior of the Congestion Avoidance Algorithm," Computer Communications Review, vol. 27, no. 3, July 1997.
[32] T. H. Henderson *et al.*, "On Improving the Fairness of TCP Congestion Avoidance," IEEE Globecomm conference, Sydney, pp. 539-544, 1998.
[33] P. Sarolahti *et al.*, "Congestion Control in Linux TCP," Proceedings of 2002 USENIX Annual Technical Conference, Freenix Track, pp. 49–62, Monterey, CA, June 2002.
[34] V. Paxson *et al.*, "Computing TCP's Retransmission Timer," Nov. 2000, Internet RFC 2988.
[35] R. Ludwig *et al.*, "The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions," ACM Computer Communications Review, vol. 30, no. 1, January 2000.
[36] P. Sarolahti *et al.*, "F-RTO: an Enhanced Recovery Algorithm for TCP Retransmission Timeouts," Computer Communication Review, vol. 33, no. 2, pp. 51--63, 2003.
[37] http://dast.nlanr.net/Projects/Iperf/.
[38] http://www.tcptrace.org/.
[39] W. Wu *et al*., "The Performance Analysis of Linux Networking–Packet Receiving," to appear in Computer Communications, Elsevier, DOI: 10.1016/j.comcom.2006.11.001.
[40] R. Love, "Interactive Kernel Performance – Kernel Performance in Desktop and Real-time Applications," Proceedings of the Linux Symposium, Ottawa, Onario, Canada, July 2003.
[41] A. Kulyavtsev *et al.*, "Resilient dCache: Replicating Files for Integrity and Availability," Proceedings of Computing in High Energy Physics (CHEP), Mumbai, India, 2006.
[42] A. Silberschatc *et al.*, Operating System Concepts, Seventh Edition, John Wiley & Sons, ISBN: 0471694665, 2004.