LyraNET: A Zero-Copy TCP/IP Protocol Stack for Embedded Operating Systems

Yun-Chen Li Mei-Ling Chiang
Department of Information Management
National Chi-Nan University, Puli, Taiwan, R.O.C.
s1213526@ncnu.edu.tw, joanna@ncnu.edu.tw

Abstract

Embedded systems are usually resource limited in terms of processing power, memory, and power consumption, thus embedded TCP/IP should be designed to make the best use of limited resources. Applying zero-copy mechanism can reduce memory usage and CPU processing time for data transmission. Power consumption can be reduced as well.

In this paper, we present the design and implementation of zero-copy mechanism in the target embedded TCP/IP component, LyraNET, which is derived from Linux TCP/IP codes and remodeled as a reusable software component that is independent from operating systems and hardware. Performance evaluation shows that TCP/IP protocol processing overhead can be significantly decreased by 23-56.22%. Besides, object code size of this network component is only 78% of the size of the original Linux TCP/IP stack. The experience of this study can serve as the reference for embedding Linux TCP/IP stack into a target system and improving the transmission efficiency of Linux TCP/IP by zero-copy implementation.

1. Introduction

As the explosion of Internet, adding Internet connectivity is required for embedded systems [5]. TCP/IP protocol is the core technology for this connectivity. In order to suit for resource-limited embedded devices, some commercial products implemented TCP/IP protocol stack from scratch for embedded systems with the aims to reduce code size and CPU processing overhead. Their codes are not freely obtainable. Since Linux provides open source codes, besides, it is popular and has the advantages of stability, reliability, high performance, and well documentation, these advantages let making use of the existing open source codes and integrating Linux TCP/IP protocol stack [6] into a target operating system become a cost-effective way.

However, because Linux is a monolithic kernel, Linux TCP/IP stack is not a separate component that has closely relationship and interaction with other Linux kernel functions such as file system, device driver, and kernel core. This adds the difficulties in reusing the Linux TCP/IP stack in a target system.

Besides, straight porting of the Linux TCP/IP protocol stack into a target operating system is also not the best implementation for the particular needs of an embedded system. Especially, embedded systems are usually resource limited in terms of processing power, memory, and power consumption. For example, data transmission of Linux TCP/IP protocol codes are suitable for general-purpose operating systems in the common resource-abundant desktop computers. Transmitted data is always copied from user buffer to kernel buffer, then sent from kernel buffer to network interface card (NIC). Received data is brought from NIC to kernel network buffer, then copied from kernel network buffer to user buffer. These data copy operations need CPU processing time and add to power consumption. Therefore, TCP/IP implementation for embedded systems should minimize the amount of data copying in order to reduce power consumption and provide efficient response.

Zero-copy [2] is a mechanism in which data from network card is directly received in the user buffer and data from user buffer is directly sent to network card. No data copying between user buffers and kernel buffers is needed. Zero-copy implementation requires virtual memory operations such as page remapping and hardware supported devices such as DMA controller. Data consistency of TCP/IP transmission must be ensured as well. Besides, because virtual memory operations and DMA are needed to implement zero-copy, memory buffers that are used to receive or send data via network must be constrained. For devices do not support DMA operations, data copying from/to network card to/from user buffers is still need.

For reusing Linux TCP/IP codes, we have extracted TCP/IP protocol stack from Linux in our previous study [4]. It is then implemented as a software

Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)
1533-2306/05 $20.00 © 2005 **IEEE**

component that is independent from operating systems and hardware, called LyraNET. Based on the component design principle [1], the advantages of modularity, reconfigurability, component replacement and reuse can be obtained. To implement the TCP/IP stack as a self-contained component requires modifying Linux TCP/IP codes to separate them from other kernel functions and implementing kernel support modules in the target operating system for integrating Linux TCP/IP protocols.

For adapting LyraNET into embedded systems, in order to reduce protocol processing overhead, memory usage, and power consumption, in this paper, we focus on applying zero-copy mechanism to reduce the data copying operations in TCP/IP transmission by passing the address of user data buffer when sending data to network, and by page remapping when receiving data from network. Besides, NIC drivers need modifications to incorporate zero-copy mechanism. After integrating LyraNET with copy elimination into LyraOS [3,8], a component-based embedded operating system, performance evaluation shows that TCP/IP protocol processing overhead can be decreased by 23-56.22%.

2. LyraOS and LyraNET

LyraOS [3,8] is a component-based operating system which aims at serving as a research vehicle for operating systems and providing a set of well-designed and clear-interface system software components that are ready for Internet PC, hand-held PC, embedded systems, etc. It was implemented mostly in C++ and few assembly codes. It is designed to abstract the hardware resources of computer systems, such that low-level machine dependent layer is clear cut from higher-level system semantics. Thus, it can be easily ported to different hardware architectures [3].

Figure 1 shows LyraOS system architecture. Each system component is complete separate, self-contained, and highly modular. Besides being light-weight system software, it is a time-sharing multi-threaded microkernel. Threads can be dynamically created and deleted, and thread priorities can be dynamically changed. It provides a preemptive prioritized scheduling and supports various mechanisms for passing signals, semaphores, and messages between threads. On top of the microkernel, a micro window component with Windows OS look and feel is provided. Besides, the LyraFILE component, a light-weight VFAT-based file system, supports both RAM-based and disk-based storages. Especially, LyraOS provides the Linux device driver emulation environment to make use of Linux device drivers. Under this environment, Linux device driver codes can be integrated into LyraOS without modification.

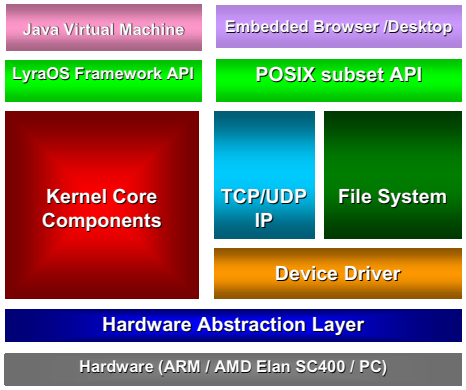


Figure 1. LyraOS system architecture

The LyraNET [4] component is a TCP/IP protocol stack derived from Linux TCP/IP codes [6]. We made the most use of Linux open source codes mainly to reduce our development effort. We then remodeled it as a reusable software component that is independent from operating systems and hardware. Our work mainly includes remodeling Linux TCP/IP stack to separate it from file systems, implementing wrappers for kernel and device independence, and providing wrapper for compatible socket interfaces.

3. Adaptation for embedded systems

To adapt LyraNET component for resource-limited embedded systems, we focus on reducing memory usage and CPU processing overhead, with the aim to reduce power consumption as well. In our network buffer management, pre-allocated buffers are used rather than allocating them at run time if buffers are needed. Copy elimination is implemented in LyraNET since the original Linux TCP/IP protocol stack [6] does not implement zero-copy mechanism. Our work includes remodeling TCP/IP protocol procedure, modifying TCP/IP protocol codes and NIC driver codes, and adding related kernel support functions.

3.1. Issues and difficulties

For zero-copy implementation, different procedures of receiving data and sending data add the difficulties to eliminate data copying overheads. When system sends data to network by a TCP/IP connection, system must establish a connection with destination host. When the connection is established, then system sends

data to network. At this time, the address of the data to be transmitted is known in advance. If zero-copy mechanism is applied, the user data can be directly sent to NIC. Whereas, when a packet is received by NIC, only after the protocol processing is performed then the system can find out which connection the packet belongs and how to process the payload of packet.

For zero copying, the payload of a packet should be allocated at right place when NIC receives incoming data. The direct way is to modify NIC drivers to send packets to buffers after the completion of protocol processing, and NIC should be designed to contain a large memory for storing unprocessed packets. Because of the lack of such devices, zero copying must be accomplished by virtual memory (VM) operations in our system.

Besides the modification of TCP/IP protocol for zero copying, NIC drivers may also need modification for incorporating zero-copy handling. Especially, NIC drivers are hardware dependent and should be implemented in different ways for network cards with or without DMA support. The modification of NIC drivers should as efficient as possible and not to degrade the total performance improvement.

3.2. Remodeling data processing flow of TCP/IP transmission

The original data transmission in TCP/IP protocol must copy data to/from kernel buffer from/to user buffer when sending/receiving a packet. We modified the original TCP/IP transmission to eliminate the data copying overhead. As shown in Figure 2, user data is directly written to NIC when data is sent to network. For the unknown destination of incoming packets and the lack of a large memory on NIC, incoming packets should be received in host memory immediately. After protocol processing is performed, destination of a packet can be known and then data can be "transmitted" to user buffer by VM operations.

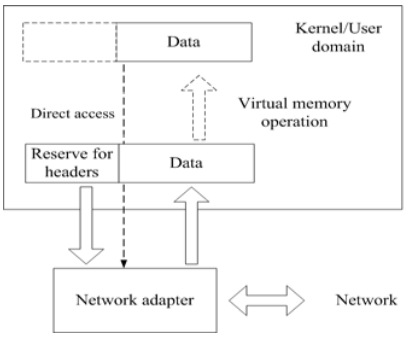


Figure 2. Dataflow for LyraNET with zero copy

3.3. Implementation of copy elimination in LyraNET

In Linux TCP/IP, the sk_buff buffer is used to manage individual packet and the maximal payload is 1460 bytes in Ethernet network. In the original Linux TCP/IP codes, user sent data is copied into one or several sk_buff buffers according to the data length. A sk_buff buffer that still has space left after being copied data into may be filled with data again.

In the implementation of copy elimination, we modify the *sk_buff* structure to avoid data copying as shown in Figure 3. The sk_buff structure is added in an array with two elements, named *dataseg*, to record addresses for data without copying. Each *dataseg* is defined with two variables, *ptr* and *len*, which record the memory addresses to be sent to network and length of data that is not copied into sk_buff buffer. The variables, *follow_data* and *follow_data_len* are added to record the address of data that is copied into a sk_buff buffer and its corresponding length. These new variables are used to avoid most of data copying by recording data address and data length.

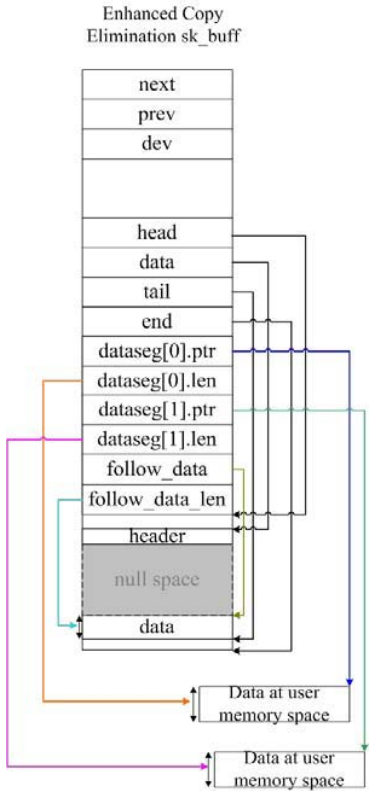


Figure 3. New sk_buff structure in copy elimination

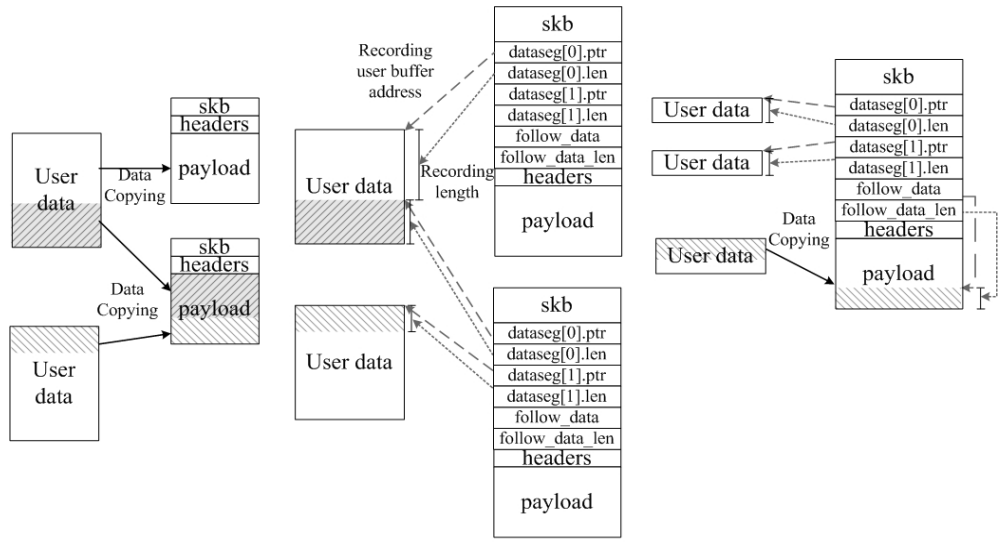


Figure 4. Data processing in original Linux TCP/IP and in LyraNET with copy elimination

When user sends data to network, a sk_buff buffer records user data in *dataseg*, as shown in Figure 4. When a sk_buff buffer is allocated to carry user buffer, user buffer address is recorded in *dataseg[0].ptr* and buffer length that is carried is recorded in *dataseg[0].len*. If first element of *dataseg* is recorded with user buffer address and the data size that sk_buff buffer carries is less then 1460 bytes, the second element of dataseg is used to record user buffer address and user buffer length that sk_buff buffer can contain. Only when the data size that sk_buff buffer carries is less then 1460 bytes and two elements of *dataseg* are recorded with user data addresses, user data is copied into sk_buff buffer. *follow_data* records the address of copied data in sk_buff and follow_data_len records the accumulated data length after data is copied into sk_buff buffer. A NIC driver should write protocol headers in sk_buff buffer into NIC, then write data that is recorded by *dataseg*, and then write data that is copied into sk_buff buffer if needed.

When receiving a packet, the modified NIC driver writes data of the incoming packet in a pre-allocated memory space that is 4096 bytes. *dev_alloc_skb()* is modified to allocate a sk_buff buffer with a page size. After the completion of protocol processing, the modified TCP/IP protocol does *page remapping* instead of copying data into user memory buffer. For the page remapping, users must allocate memory through a special system call to allocate page-aligned user buffers, and Copy on Write (COW) mechanism is implemented to maintain the data consistency.

3.4. Modifications of NIC drivers

NIC drivers should be modified to work with Copy Elimination. In PIO NIC, data is transmitted from/to NIC to/from host memory by CPU, whereas, in DMA NIC, data is transmitted from/to NIC to/from host memory by DMA device. Though DMA device can eliminate one data copying from the viewpoint of CPU, the use of DMA device limits zero copy in some way.

In LyraOS on x86 platform, a continuous virtual memory space may be mapped to several physical memory page frames that are not contiguous. If data size is larger than 4096 bytes, DMA controller may fail to transmit data when the segment of data crosses the page boundary. Therefore, data segment that is not copied into sk_buff buffer should be checked for crossing page boundary. For DMA NIC driver, checking for page boundary is done when user data is separated into sk_buff buffer. When data segment crosses page boundary, the data segment is taken as two data segments and stored into sk_buff buffer. So data segment stored in *dataseg* is not needed to check if it crosses page boundary.

3.5. Added kernel support functions

Because page remapping is needed to implement Copy Elimination, we have implemented related kernel support functions, COW mechanism, and page fault recovery routines. Besides, a specific function is provided for users to allocate page-aligned buffers.

4. Performance evaluation

This section presents the performance evaluation of LyraNET with Copy Elimination after being integrated into LyraOS. Table 1 shows our experimental platform that we use to simulate an embedded system, in which two computers are connected in a private network to avoid the affection of external network traffic. The ttcp [7] benchmark is used and the processing times for protocol processing and network driver operations are measured. The total data length is set to 26,280K bytes for PIO NIC and 131,400K bytes for DMA NIC.

Table 1. Experimental platform

	LyraOS	Linux 2
	Linux 1	
CPU	Pentium 200 MHz	Pentium 166 MHz
RAM	16MB (32MB for Linux 2.4.20)	32MB
OS	LyraOS	Linux 2.0.37(for PIO)
	Linux 2.0.37(for PIO)	Linux 2.4.20(for DMA)
	Linux 2.4.20(for DMA)	
PIO NIC	NE2000	NE2000
DMA NIC	3com 3c905cx-TX-M	Intel 82550

4.1. Comparison of object code size

Table 2 shows that the object code size of LyraNET with Copy Elimination is 78% of the size of Linux TCP/IP Stack. Adding Copy Elimination mechanism in LyraNET only increases 1.1-1.7% of object code size.

Table 2. Code size comparison

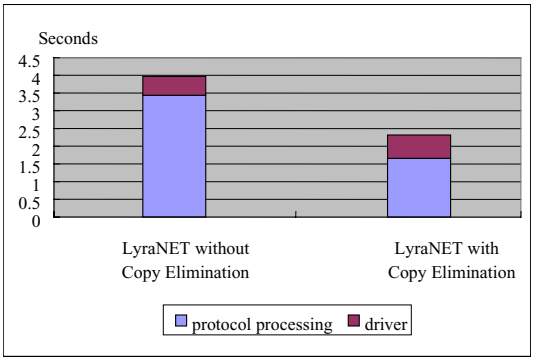
	Object Code Size (bytes)
Linux 2.0.37 TCP/IP stack	116,892
LyraNET without Copy Elimination	89,760
LyraNET with Copy Elimination (PIO NIC)	91,241
LyraNET with Copy Elimination (DMA NIC)	90,760

4.2. Performance of sending data

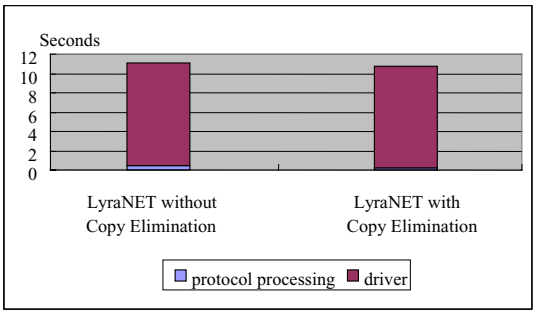
We send data with the size several times of 1460 bytes. Figure 5 shows that protocol performance improvement is from 51.34-56.22% when Copy Elimination is applied in LyraNET.

When DMA NIC is used, though driver processing time of Copy Elimination is increased, total processing time is still decreased by 27.7-50%. Because of the fast speed of DMA controller, driver processing time is efficient and does not dominate the total processing time when DMA NIC is used. This concludes that Copy Elimination is beneficial when data copying dominates the total processing time.

When PIO NIC is used, the driver processing time becomes an extremely large portion of total processing time due to the characteristic of PIO. Total processing time of Copy Elimination is still decreased slightly because protocol processing time is decreased.



(a) Sending data by DMA NIC



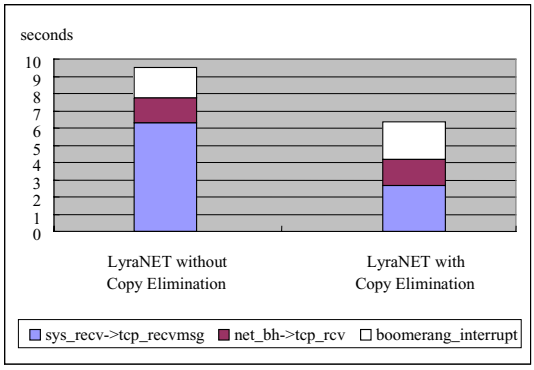
(b) Sending data by PIO NIC

Figure 5. Processing time for sending data

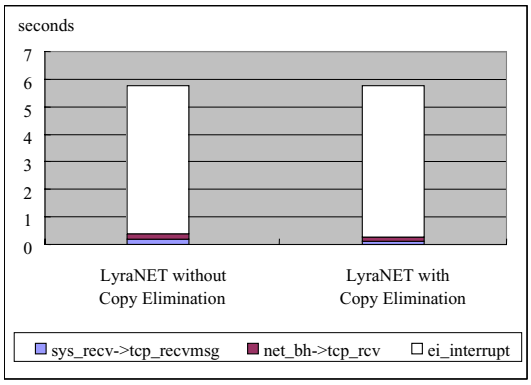
4.3. Performance of receiving data

We measure three parts of processing time: NIC driver operation (i.e. *ei_interrupt()* or *boomerang_interrupt()*), main protocol codes of receiving data (i.e. from *net_bh()* to *tcp_rcv()*), and the codes of system call processing (i.e. from *sys_recv()* to *tcp_recvmsg()*). In Linux source codes, incoming packets from NIC are received in NIC interrupt service routine (ISR), then this ISR marks *NET_BH* to activate bottom half handling, i.e. *net_bh()*. Most of the receiving protocol processing is completed in the control flow from *net_bh()* to *tcp_rcv()*. Then *tcp_rcv()*

calls *tcp_data()* to wake up the thread waiting for the data. The waiting thread that slept in *tcp_recvmsg()* is waken up to copy data while the thread previously calls *sys_recv()* to receive data.



(a) Receiving data by DMA NIC



(b) Receiving data by PIO NIC

Figure 6. Processing time for receiving data

Figure 6(a) shows the processing time when system receives data by DMA NIC. The ttcp is setup to send 1460-byte data 102,400 times to LyraNET. The results show that the processing time of data copying (i.e from *sys_recv()* to *tcp_recvmsg()*) in origin TCP/IP stack is the largest part of total processing time. With Copy Elimination, the processing time of data copying is decreased greatly. Though *boomerang_interrupt()* is not modified in Copy Elimination, however, page remapping would incur TLB flushing, which in turn would degrade performance of DMA driver. In protocol processing part, the difference of the processing time from *netbh()* to *tcp_rcv()* in LyraNET with and without Copy Elimination is insignificant.

Figure 6(b) shows the processing time when system receives data by PIO NIC. The ttcp is setup to send 1460-byte data 8192 times to LyraNET. The results show that receiving data from NIC is the main bottleneck. Without support of fast device such as DMA controller, PIO NIC relies on CPU to receive data to host memory. Though we greatly reduce the processing time of data copying in Copy Elimination, however, driver processing dominates the total processing time, which causes performance improvement insignificant.

5. Conclusions

We have reused and remodeled Linux TCP/IP stack to be a software component called LyraNET that is independent from operating systems and hardware. For the adaptation into resource-limited environments, we develop Copy Elimination in LyraNET to reduce protocol processing overhead and reduce memory usage. Performance evaluation shows that protocol processing time can be reduced by 51.34-56.22% in sending data and by 23-46% in receiving data. Adding Copy Elimination mechanism only increases 1.1-1.7% of object code size. To sum up, the success and the experience of our work can serve as the reference for embedded Linux TCP/IP stack into a target system requiring network connectivity. Besides, our zero copy implementation can also help the work of enhancing the transmission efficiency of Linux TCP/IP stack.

6. References

[1] J. Bruno, J. Brustoloni, E. Grabber, A. Silberschatz, and C. Small, "Pebble: A Component Based Operating System for Embedded Applications", In *Proceedings of 3rd Symposium on Operating Systems Design and Implementation*, USENIX, February 1999.

[2] J. C. Brustoloni and P. Steenkiste, "Effects of Buffering Semantics on I/O Performance", *Proceedings of 2nd Symposium on Operating Systems Design and Implementation*, pages 277–291, USENIX, Oct. 1996.

[3] Z. Y. Cheng, M. L. Chiang, and R. C. Chang, "A component based operating system for resource limited embedded devices", *IEEE International Symposium on Consumer Electronics*, HongKong, Dec. 5-7, 2000.

[4] J. W. Chuang, K. S. Sew, M. L. Chiang, and R. C. Chang, "Integration of Linux Communication Stacks into Embedded Operating Systems", *International Computer Symposium*, December 6-8, 2000.

[5] T. Herbert, "Embedding TCP/IP", http://www.embedded.com/2000/0001/0001ia1.htm.

[6] S. T. Satchell and H. B. J. Clifford, *Linux IP Stacks Commentary*, Coriolis Group Books, 2000.

[7] ttcp, http://www.clarkson.edu/projects/itl/HOWTOS/PCATTCP-jnm-20011113.htm .

[8] C. W. Yang, C. H. Lee, and R. C. Chang, "Lyra: A System Framework in Supporting Multimedia Applications", *IEEE International Conference on Multimedia Computing and Systems'99*, Florence, Italy, June 1999.