Real-Time Syst (2006) 34:5–18
DOI 10.1007/s11241-006-6879-7

LyraNET: A zero-copy TCP/IP protocol stack for embedded systems*

Mei-Ling Chiang · Yun-Chen Li

Published online: 21 April 2006
© Springer Science + Business Media, LLC 2006

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 23–63%. Besides, object code size of this network component is only 77.64% 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 that requires network connectivity and improving the transmission efficiency of Linux TCP/IP by zero-copy implementation.

Keywords Embedded TCP/IP · Zero-copy · Embedded operating systems · Linux

1. Introduction

As the explosion of Internet, adding Internet connectivity is required for embedded systems (http://www.embedded.com/internet/0001/0001ia1.htm). TCP/IP protocol (Wright and

* This paper is an extended version of the paper "LyraNET: A Zero-Copy TCP/IP Protocol Stack for Embedded Operating Systems" that appeared in the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications.

M.-L. Chiang (✉) · Y.-C. Li
Department of Information Management, National Chi-Nan University,
Puli, Taiwan, R.O.C.
e-mail: joanna@ncnu.edu.tw;s1213526@ncnu.edu.tw

Stevens, 1994) is the core technology for this connectivity. In order to suit for resource-limited embedded devices, some commercial products (http://sources.redhat.com/ecos/, 2004; http://www.unicoi.com/fusion_net/fusion_tcpip.htm; Massa, 2002; http://www.ucos-ii.com/, 2004; http://www.iniche.com/nichestack.php; http://www.blunkmicro.com/tcp.htm) implemented TCP/IP protocol stack from scratch for embedded systems with the aims to reduce code size and CPU processing overhead. Most of them 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 (Satchell et al., 2000) 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 systems, device drivers, and kernel core. This adds the difficulties in reusing the Linux TCP/IP stack in a target system.

Besides the difficulties of reusing Linux TCP/IP stack, 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 is 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 network interface card 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 (Brustoloni and Steenkiste, 1996; Chase et al., 2001; Chu, 1996; Druschel and Peterson, 1993; Steenkiste, 1994, 1998) 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. 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 (Chuang et al., 2000). It is then implemented as a software component that is independent from operating systems and hardware, called LyraNET. Based on the component design principle (Bruno et al., 1999; Chen et al., 2000; Friedrich et al., 2001; Grabber et al., 1999), the advantages of modularity, reconfigurability, component replacement and reuse can be obtained. To implementing 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 (Chen, 2000; Chen et al., 2000; Yang et al., 1999), a component-based embedded operating system, performance evaluation shows that TCP/IP protocol processing overhead can be decreased by 23–63%.

The rest of this paper is organized as follow. Section 2 briefly introduces LyraOS and LyraNET component. Section 3 presents how we implement zero-copy mechanism in LyraNET to reduce TCP/IP protocol processing time and the related modifications to NIC drivers. Section 4 presents experimental results. Section 5 discusses related work, and Section 6 concludes this paper.

2. LyraOS and LyraNET

LyraOS (Chen, 2000; Chen et al., 2000; Yang et al., 1999) 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 (Chen, 2000).

Figure 1 shows system architecture of LyraOS. 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 adjusted. 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 (Huang, 2000). Besides, the LyraFILE component (Chiang and Lo, 2006; Ting et al., 2002), a light-weight VFAT-based file system, supports both RAM-based and disk-based storages. Especially, LyraOS provides the Linux device driver emulation environment (Chen, 2004; Yang, 1998; Yang et al., 1998) to make use of Linux device drivers. Under this emulation environment, Linux device driver codes can be integrated into LyraOS without modification.

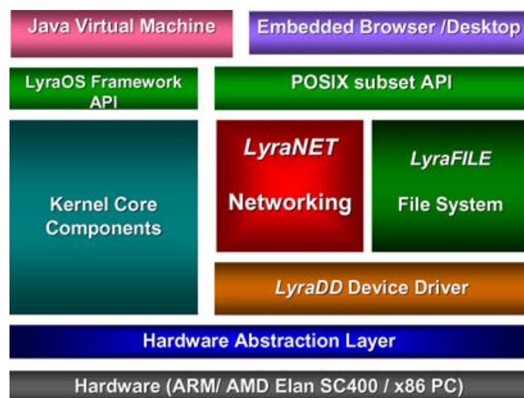


Fig. 1 LyraOS system architecture

The LyraNET (Chuang et al., 2000) component is a TCP/IP protocol stack derived from Linux TCP/IP codes (Rusling, 2002). 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 (Rusling, 2002) is not implemented with 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 first 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 already known. 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 can the system find out which connection the packet belongs to 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 driver to send packets to application 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 modifications 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 be as efficient as possible and not 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, as shown in Figure 2(a). We modified the original TCP/IP transmission to eliminate the data copying overhead. 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 virtual memory operations. The data flow of modified TCP/IP transmission is shown in Figure 2(b).

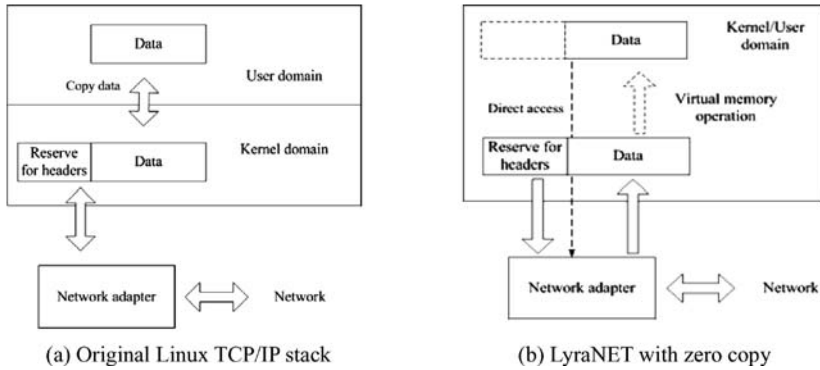


Fig. 2 Dataflow of TCP/IP protocol processing: (a) Original Linux TCP/IP stack and (b) LyraNET with zero copy

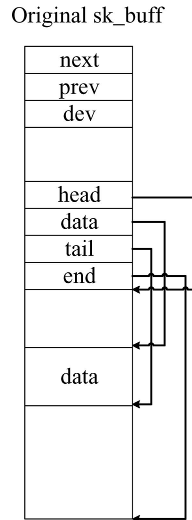


Fig. 3 Original *sk_buff* structure in Linux TCP/IP

3.3. Implementation of copy elimination in LyraNET

In Linux TCP/IP, the *sk_buff* buffer as shown in Figure 3 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 4. 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.

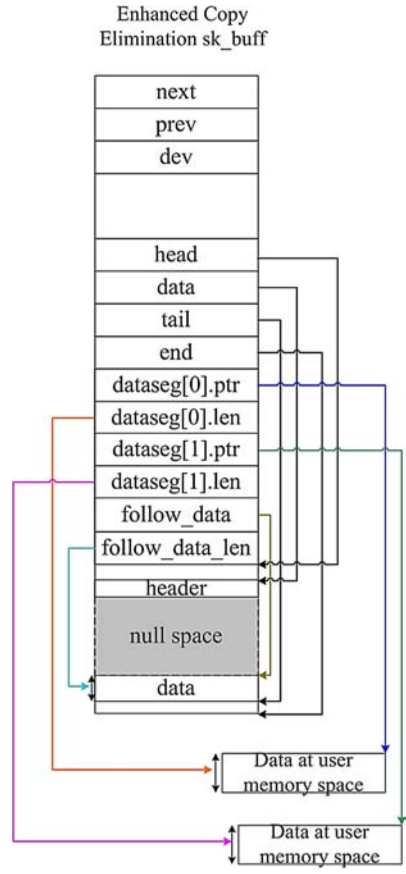


Fig. 4 New *sk_buff* structure in Copy Elimination

When user sends data to network, a *sk_buff* buffer records user data in *dataseg*, as shown in Figure 5. 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 the first element of *dataseg* is recorded with user buffer address and the data size that the *sk_buff* buffer carries is less than 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 the *sk_buff* buffer carries is less than 1460 bytes and two elements of *dataseg* are recorded with user data addresses, user data is copied into *sk_buff* buffer. The *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 first write protocol headers in *sk_buff* buffer into NIC, then write data that is recorded by *dataseg*, and finally write data that is copied into *sk_buff* buffer if needed.

When receiving an incoming packet, the modified NIC driver writes data of the incoming packet in a pre-allocated memory space that is 4096 bytes. The *dev_alloc_skb()* function 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 operation, users must allocate memory through

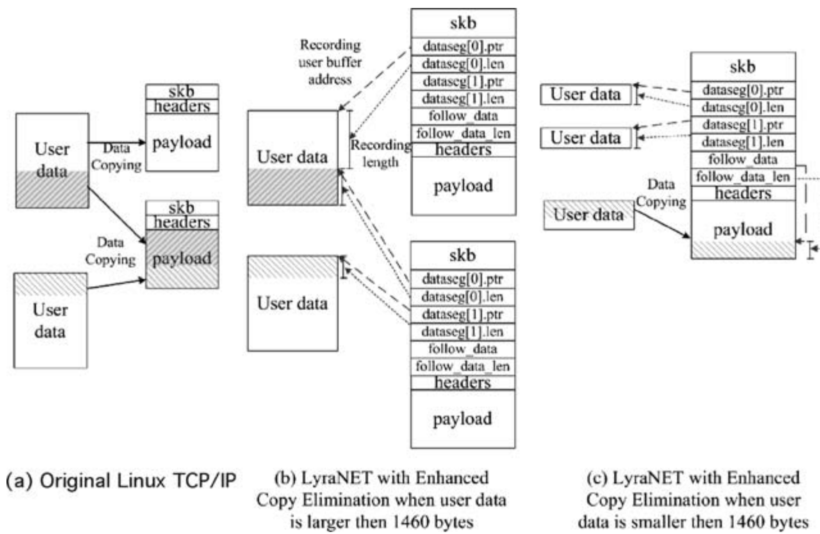


Fig. 5 Data processing in original Linux TCP/IP and in LyraNET with Copy Elimination

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 for copy elimination

In order to avoid data copying, the *sk_buff* data structure is modified to achieve Copy Elimination. For this modification of *sk_buff*, NIC drivers should be modified to work with Copy Elimination. According to the transmission mode, NICs are divided into PIO NIC and DMA NIC. 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.

When a NIC driver informs DMA controller to start to send/receive data, the command often includes data address and data length. Because a DMA device uses bus address whereas a program uses virtual address, the virtual address should be transformed into bus address.

In LyraOS on x86 platform, virtual memory management is page-based and the page size is 4096 bytes. 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. As for DMA NIC driver, DMA NIC driver doesn't transmit data, it communicates with DMA controller to transmit data. 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 sub-data segments and stored into *sk_buff* buffer, as shown in Figure 6. So data segment stored in *dataseg* is not needed to check if it crosses page boundary. Though DMA device is efficient for mass data transmission, the related modification of DMA NIC driver would increase the processing time.

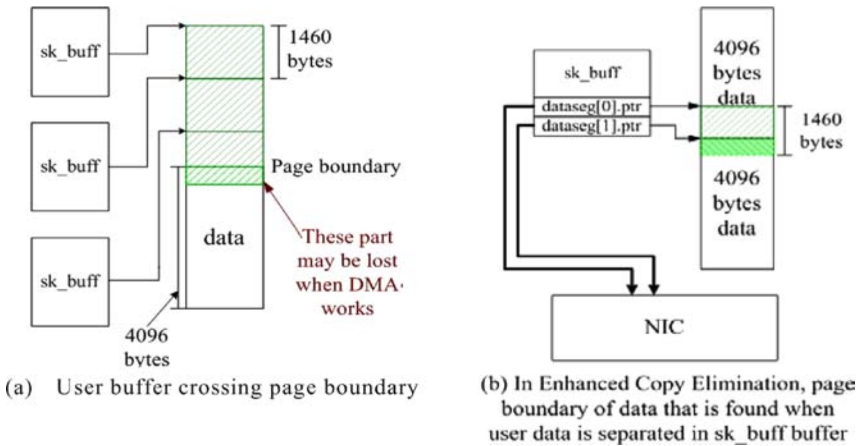


Fig. 6 Handling for user buffer crossing page boundary

3.5. Added related 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.

The pageremap() is implemented for page remapping. The writeprotect() and unwriteprotect() are provided for write permission control, in order to implement COW. The PktMemoryAlloc() is provided for users to allocate page-aligned buffers. The AllocPhysicalPage() is implemented for page fault recovery of COW. AllocPhysicalPage() allocates a new physical memory page to the virtual memory that causes page fault in page fault routine.

In implementation of page remapping, two virtual memory pages are mapped to the same physical memory page. While data is under transmission, write permission control is needed to refrain users from modifying the content of a physical memory page. If a virtual memory that is remapped to a physical memory page is modified with new content, a page fault would occur because of the wrong write permission of PTE. In page fault routine, error code is checked for the reason of page fault. If the error code represents wrong write permission of PTE, then system invokes AllocPhysicalPage() to allocate a physical memory with read-write permission for the virtual memory that is without write permission and then system recovers from page fault.

Fig. 7 PktMemoryAlloc() returns the allocated memory address which has been offset

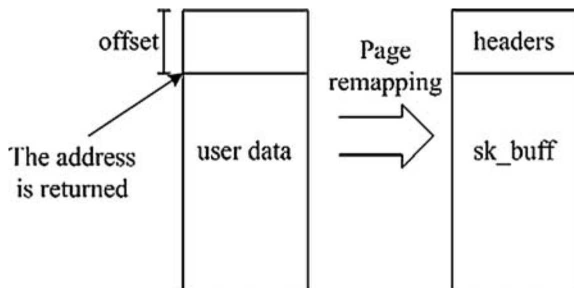


Table 1 Code size comparison

	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

PktMemoryAlloc() is implemented to allocate a page-aligned memory for receiving data from network in Copy Elimination. As shown in Figure 7, PktMemoryAlloc() allocates a page-sized memory, and returns the address which has been offset. Offset is needed to be set in order to avoid accessing protocol headers in *sk_buff* buffer. This is because the user memory created by PktMemoryAlloc() is remapped to the *sk_buff* buffer such that the allocated user buffer and the *sk_buff* buffer share the same physical memory. Since protocol headers in *sk_buff* buffer are not needed for user, data in *sk_buff* buffer that is really needed for user is behind these protocol headers.

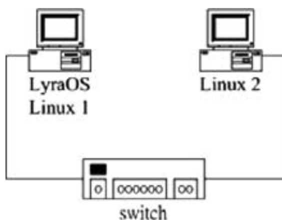
4. Performance evaluation

This section presents the performance evaluation of LyraLNET with Copy Elimination after being integrated into LyraOS. Figure 8 shows our experimental environment and the 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.

We use the popular ttcp (http://www.clarkson.edu/projects/itl/HOWTOS/PCATTCP-jnm-20011113.htm) benchmark to perform the experiments and vary the amount of data in transmission in measuring the processing time for protocol processing and network driver operations. We also vary the transmission times and data length to control the total data length in transmission. The PIO NIC is a 10BASE-T NIC and DMA NIC is a 100BASE-TX NIC in our experimental platform. The total data length is set to 26,280 K bytes for PIO NIC and 131,400 K bytes for DMA NIC.

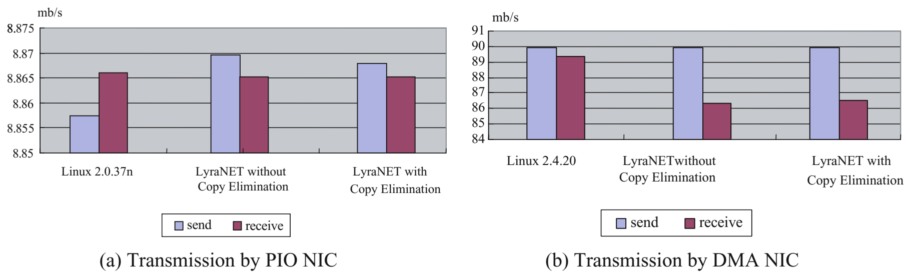
4.1. Comparison of object code size

Table 1 shows that the object code size of LyraNET with Copy Elimination is 77.64% of the size of Linux TCP/IP Stack. Adding Copy Elimination mechanism in LyraNET only increases 1.11–1.65% of object code size.



	LyraOS Linux 1	Linux 2
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

Fig. 8 Experimental environment



(a) Transmission by PIO NIC (b) Transmission by DMA NIC

Fig. 9 Data transmission time

4.2. Data transmission time

This section evaluates the performance results of transferring 2920 times of 8 K-byte packets for ttcp benchmark. In order to compare the performance difference under different systems, the same evaluation was conducted under Linux, LyraNET without Copy Elimination, and LyraNET with Copy Elimination. Figure 9(a) shows the experimental results of data transfer speed in PIO NIC. The speed of sending data of Linux is slightly slower than LyraNET, but the speed of receiving data is almost the same for these systems. Figure 9(b) shows that these systems all have the equivalent speed of sending data when DMA NIC is used. Whereas, speed of receiving data in Linux is 3.4% faster than that in LyraNET. This is possible due to the differences of TCP/IP protocol stack in Linux 2.4.20 (http://www.tldp.org/LDP/lki/index.html, 2002) and in Linux 2.0.33 (Rusling, 2002) which is the kernel LyraNET is derived from. Though LyraOS and Linux have equivalent performance, whereas, LyraNET is only 77.64% of the size of the Linux TCP/IP stack.

4.3. Performance of sending data

In the experiments of sending data, we evaluate the effect of the different length of transmitted data on the CPU processing time for TCP/IP protocol codes and NIC driver. We send data with the size of several times of 1460 bytes and 1024 bytes. Figure 10 shows the processing time of different parts for sending data. Total processing time of LyraNET with Copy Elimination is less than that of LyraNET without Copy Elimination. Especially, protocol performance improvement is from 49–63% 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. In fact, DMA is better used for network systems with larger size of MTU.

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.

4.4. Performance of receiving data

Receiving data is constrained by special VM operations, so we conduct the experiment of receiving 1460 bytes of data. We measure three parts of processing time: NIC driver

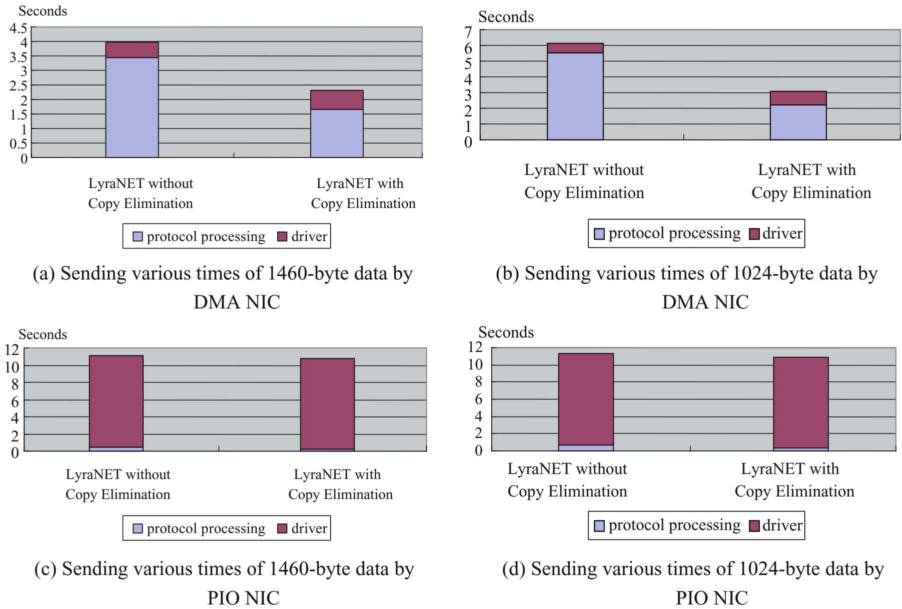
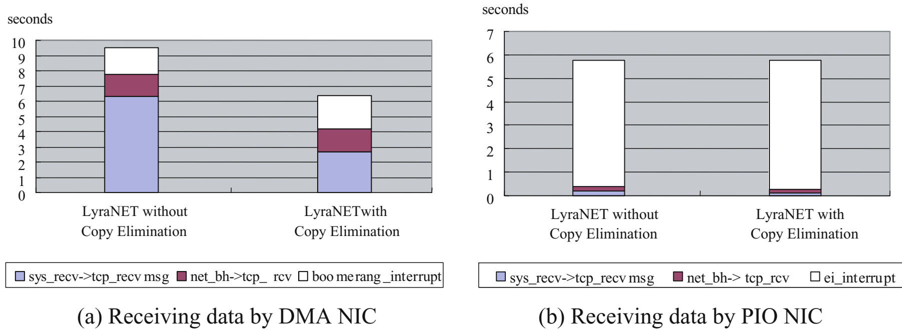


Fig. 10 Processing time comparison for sending data

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.

Figure 11(a) shows the processing time when system receives data by DMA NIC. The ttcp is set up 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 original 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 11(b) shows the processing time when system receives data by PIO NIC. The ttcp is set up 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 copy received 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.



(a) Receiving data by DMA NIC (b) Receiving data by PIO NIC

Fig. 11 Processing time comparison for receiving data

5. Related work

Most of TCP/IP protocol stacks for embedded systems are designed to reduce code size and CPU processing overhead in order to suit for resource-limited embedded devices. Some embedded TCP/IP stacks are also implemented with the feature of zero copy. For example, lwIP (http://savannah.nongnu.org/projects/lwip/, 2004) is a small independent implementation of the TCP/IP protocol suite. lwIP is implemented as a thread that can work in a standalone environment. It is implemented for the reduction of the RAM usage while still having full scale TCP. In lwIP, if the data is stored in ROM, it is sent to kernel by recording the data address instead of allocating a kernel buffer and copying data into this newly allocated buffer. However, the mechanism of avoiding data copying in lwIP is only used in sending data to network, and is not for receiving packets from network yet. This indicates that minimizing data copying for receiving data is more complicated.

Several commercial products implement embedded TCP/IP from scratch and are all designed with the aim to be small, portable, high performance, and ROMable. Especially, they are not free and not open source. TargetTCP (http://www.blunkmicro.com/tcp.htm) is developed by Blunk Microsystems on TargetOS™ (http://www.blunkmicro.com/os.htm). When the Berkeley sockets API is used, one data copying of application data is performed. Whereas, a zero-copy API is provided to eliminate the data copying for socket API. Fusion Embedded TCP/IP Stack protocol suite (http://www.unicoi.com/fusion_net/fusion_tcpip.htm) developed by Unicoi Systems also provides BSD sockets compatible support and zero-copy for UDP transmission. It is used in variety of backbone networking products, such as bridges, routers, and switches. NicheStack (http://www.iniche.com/nichestack.php) also implements zero copy mechanism. Besides, NicheStack is designed to be portable to standard operating system environments or to operate under no operating system support. μC/TCP-IP (http://www.ucos-ii.com/, 2004) is another TCP/IP protocol stack designed for embedded systems. Zero copy buffer management for high efficiency is also implemented.

For general-purpose computer design, the avoidance of data copying in TCP/IP transmission is designed to promote the transmission speed. Most researches often depend on virtual memory operations such as page remapping (Brustoloni and Steenkiste, 1996; Chase et al., 2001; Chu, 1996; Druschel and Peterson, 1993; Steenkiste, 1994, 1998) and copy on write mechanism for copy elimination. Some researches use special hardware support such as DMA, early demultiplexing (Steenkiste, 1998), outboard buffering

(Brustoloni and Steenkiste, 1996; Steenkiste, 1994), or hardware checksumming (Chase et al., 2001) to achieve zero copy. The differences of data copying avoidance in embedded systems and general-purpose computers are network architecture and NIC used. ATM network adapters are often used in most researches for zero-copy TCP/IP in general-purpose computers, and the MTU of ATM network is larger than the MTU of Ethernet network. Their experiments all show that minimizing data copying significantly increases system performance.

6. 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.

After integrating the LyraNET into our target embedded operating system, LyraOS, performance evaluation shows that when Copy Elimination is applied in LyraNET, protocol processing time can be reduced by 49–63% in sending data and by 23–46.16% in receiving data. Adding Copy Elimination mechanism only increases 1.11–1.65% of object code size.

To sum up, the success and the experience of our work can serve as the reference for embedding 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.

Acknowledgments We would like to thank Professor Ruei-Chuan Chang for providing his precious support and abundant research resource in the LyraOS development. We also thank Jer-Wei Chuang and Kim-Seng Sew for their effort in the implementation of the base LyraNET. This research was supported in part by the National Science Council of the Republic of China under grant No. NSC93-2213-E-260-021.

References

Bruno, J., Brustoloni, J., Grabber, E., Silberschatz, A., and Small, C. 1999. Pebble: A component based operating system for embedded applications. In *Proceedings of 3rd Symposium on Operating Systems Design and Implementation*, USENIX.

Brustoloni, J. C. and Steenkiste, P. 1996. Effects of buffering semantics on I/O performance. In *Proceedings 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, USENIX, pp. 277–291.

Chase, J., Gallatin, A., and Yocum, K. 2001. End-system optimizations for high-speed TCP. *IEEE Communications*, 39(4):68–74.

Chen, C.-H. 2004. LyraDD: Design and implementation of the device driver model for embedded systems, master thesis, Department of Information Management, National Chi-Nan University.

Chen, Z. Y. 2000. A component based embedded operating system, master thesis, Department of Information and Computer Science, National Chiao Tung University.

Chen, Z. Y., Chiang, M. L., and Chang, R. C. 2000. A component based operating system for resource limited embedded devices. In *IEEE International Symposium on Consumer Electronics* (ISCE'2000), HongKong.

Chiang, M.-L. and Lo, C.-R. 2006. LyraFILE: A component-based VFAT file system for embedded systems. To Appear in International Journal of Embedded Systems, Issue 1.

Chu, H. K. J. 1996. Zero-Copy TCP in Solaris. In *Proceedings of the USENIX 1996 Annual Technical Conference*, SanDiego, California.

Chuang, J.-W., Sew, K.-S., Chiang, M.-L., and Chang, R.-C. 2000. Integration of linux communication stacks into embedded operating systems. *International Computer Symposium* (ICS'2000).

Druschel, P. and Peterson, L. L. 1993. Fbufs: A high-bandwidth cross-domain transfer facility. *ACM SIGOPS Operating Systems Review*, 27(5):189–202.
"eCos, http://sources.redhat.com/ecos/," 2004
Friedrich, L., Stankovic, J., Humphrey, M., Marley, M., and Haskins, J. 2001. A survey of configurable component-based operating systems for embedded applications. *IEEE Micro*, 21(3):54–68.
Fusion embedded TCP/IP stack, http://www.unicoi.com/fusion_net/fusion_tcpip.htm
Grabber, E., Small, C., Bruno, J., Brustoloni, J., and Silberschatz, A. 1999. The pebble component-based operating system. In *1999 USENIX Annual Technical Conference*.
Herbert, T. Internet Appliance Design Embedding TCP/IP, http://www.embedded.com/internet/0001/0001ia1.htm
Huang, W.-S. 2000. An implementation of a configurable window system on lyraOS. Master Thesis, Department of Computer and Information Science, National Chiao Tung University.
"Linux Kernel 2.4 Internals at http://www.tldp.org/LDP/lki/index.html," 2002
"lwIP—A Lightweight TCP/IP stack, http://savannah.nongnu.org/projects/lwip/," 2004
Massa, A. J. 2002. *Embedded Software Development with eCos*, Prentice Hall PTR.
"MicroC/OS II, at http://www.ucos-ii.com/," 2004
"NicheStack IPv4, http://www.iniche.com/nichestack.php"
Rusling, D. A. 2002. The linux kernel, http://www.tldp.org/LDP/tlk/tlk.html
Satchell, S. T., Clifford, H. B. J., and Clifford, H. 2000. Linux IP stacks commentary: guide to gaining insider's knowledge on the IP stacks of the linux code, Coriolis Group Books.
Steenkiste, P. A. 1994. A systematic approach to host interface design for high-speed networks. *ACM Computer*, 27(3):47–57.
Steenkiste, P. 1998. Design, implementation, and evaluation of a single-copy protocol stack. *Software-Practice and Experience*, 28(7):749–772.
"TargetOS, http://www.blunkmicro.com/os.htm"
"TargetTCP, http://www.blunkmicro.com/tcp.htm"
Ting, H. K., Lo, C. R., Chiang, M. L., and Chang, R. C. 2002. Adapting LINUX VFAT filesystem to embedded operating systems. *International Computer Symposium (ICS'2002)*, HwaLian, Taiwan, R.O.C.
"ttcp, http://www.clarkson.edu/projects/itl/HOWTOS/PCATTCP-jnm-20011113.htm"
Wright, G. R. and Stevens, W. R. 1994. TCP/IP Illustrated Vol. 1, 1st edition. Addison-Wesley Professional.
Yang, C. W. 1998. An integrated core-work for fast information-appliance buildup, masters thesis, Department of Information and Computer Science, National Chiao Tung University.
Yang, C.-W., Lee, P. C. H., and Chang, R. C. 1998. Reuse linux device drivers in embedded systems. In *Proceeding of the 1998 International Computer Symposium (ICS'98)*, Taiwan.
Yang, C.-W., Lee, C. H., and Chang, R. C. 1999. Lyra: A system framework in supporting multimedia applications. In *IEEE International Conference on Multimedia Computing and Systems'99*, Florence, Italy.

Mei-Ling Chiang received the B.S. degree in Management Information Science from National Chengchi University, Taipei, Taiwan, in 1989. She received the M.S. degree in 1993 and her Ph.D degree in 1999 in Computer and Information Science from National Chiao Tung University, Hsinchu, Taiwan. Now she is an Assistant Professor in the Department of Information Management at National Chi-Nan University, Puli, Taiwan. Her current research interests include operating systems, embedded systems, and clustered systems.

Yun-Chen Lee received the B.S degree in 2002 and the M.S. degree in 2005 in Information Management from National Chi-Nan University, Puli, Taiwan. He is currently a software engineer in InterVideo Digital Tech., responsible for software development of multimedia-related products.