Offloading Socket Processing for Ubiquitous Services*

SUNWOOK KIM[1], SEONGWOON KIM[1], KYOUNG PARK[2]
AND YONGWHA CHUNG[3]
[1]*Department of Cloud Computing Research*
[2]*Department of Future SW Content Technology Research*
Electronic and Telecommunications Research Institute
Daejeon, 305-350 Korea
[3]*Department of Computer and Information Science*
Korea University
Chungnam, 339-700 Korea

As digital devices with communication capability become more pervasive, the network performance of a corresponding server needs to be improved. In this paper, we present the hardware and software for offloading the socket processing in order to improve the network performance of a server. The experimental results showed that the proposed solution could improve the network performance of a typical solution significantly. Furthermore, the proposed solution can provide the binary compatibility with the BSD socket standard such that the existing network programs can be used without modification and/or recompilation.

Keywords: ubiquitous/home-network service, network I/O, TCP/IP, TOE, socket offloading

1. INTRODUCTION

Today's internet consists of broadband connections mainly for PCs and servers. However, under the ubiquitous environment, new internet will consist of networks of thousands or millions of microchips, electronic appliances, mobile devices as well as PCs and servers.

In the new internet (*i.e.*, future ubiquitous communication environment), terminal-initiated traffics will become more popularized than human-initiated traffics, like current internet communications [1]. The terminals are expected as various and huge amount of sensor nodes reporting simple presence data to the servers. Thus, the server will convert the received data into the meaningful context information which can be provided to ubiquitous services.

Generally, the packet size of the terminal-initiated traffics is smaller than that of the human-initiated traffics (*e.g.*, web browsing, file transfer, and media streaming), because the presence data is tiny information obtained from various sensors reporting location information, temperature, and so on. As the internet evolves into a ubiquitous network, a server for ubiquitous services must manage not only the human-initiated traffics, but also a lot of the terminal-initiated traffics in order to provide ubiquitous services to users. Therefore, for network performance, the server needs to be optimized in order to process

Received October 6, 2009; revised February 28, 2010; accepted June 7, 2010.
Communicated by Ren-Hung Hwang, Chung-Ming Huang, Cho-Li Wang, and Sheng-Tzong Cheng.
* This work was supported by the Industrial Strategic Technology Development Program (10035242, Development of Cloud DaaS (Desktop as a Service) System and Terminal Technology) funded by the Ministry of Knowledge Economy (MKE, Korea) and under the HNRC ITRC support program supervised by the NIPA (NIPA-2010-C1090-1011-0010).

bulk of the terminal-initiated traffics in the ubiquitous environment.

TCP/IP-the predominant protocol suite across the internet-has been implemented traditionally in software as a part of the operating system kernel. A frequently cited drawback of TCP/IP is that the data copying and TCP/IP processing overhead consumes a significant share of host CPU cycles and memory bandwidth, siphoning off system resources needed for application processing [2, 3].

In general, the TCP/IP processing overhead can be divided into two categories: per-byte-cost, primarily data-touching operations such as checksums and copies; and per-packet-cost, including TCP/IP protocol processing, interrupt overhead, buffer management overhead, socket handling, and kernel overhead [3]. To lessen some of the overhead, researchers developed several mechanisms that became common in today's TCP/IP processing.

Today's advanced kernel and high performance NIC support zero-copy, segment offloading, checksum offloading, and interrupt coalescing in order to reduce the overhead of data touching overhead [3-5]. On the other hand, several industry players announced TCP Offloading Engine(TOE) devices that offload the TCP/IP processing from a host CPU [6, 7]. However, typical TOEs are only suitable for bulk large transfers involving long-lived, few connections. The major reason of such limitation of TOE is that it only focuses on offloading the TCP/IP protocol processing and data touching overhead (per-byte-cost). Moreover, it requires another communication overhead between a TOE device and a host in order to lookup socket and TCP Control Block (TCB) [4, 5]. Also, TOE cannot eliminate per-packet-cost including the overhead of socket handling and kernel. Its specialized API and stack code cause troubles in designing or porting general network applications [4].

In this paper, we present the hardware and software for improving the network performance required for ubiquitous services. Unlike typical TOEs, we offload the socket processing in addition to the TCP/IP processing, and focus on the terminal-initiated traffics as well as the human-initiated traffics. Generally, sockets are created by every connection or link activated by a server and/or a client. Thus, the number of sockets created and managed by a protocol stack increases as the number of processes increases. This causes a considerable load for processing a network protocol in a system, thereby degrading overall performance of the system. Furthermore, our solution supports the binary compatibility with the standard socket interface such that the existing network programs can be used without modification and/or recompilation.

The rest of this paper is organized as followings. Section 2 describes the overview of the LATONA (Leading Architecture for TCP/IP Offloading and Network Acceleration). Section 3 discusses the design details of our solution for offloading the socket processing. In section 4, we discuss the experimental results, and finally section 5 presents our conclusion.

2. ARCHITECTURE OF LATONA

2.1 Kernel Architecture of LATONA

The LATONA kernel is a set of dedicated kernel modules which replace the tradi-

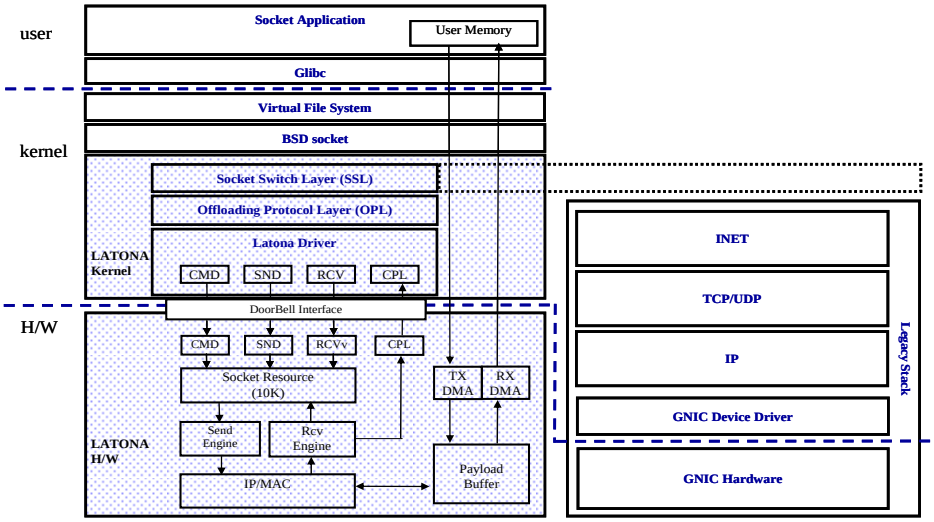


Fig. 1. The overall architecture of LATONA.

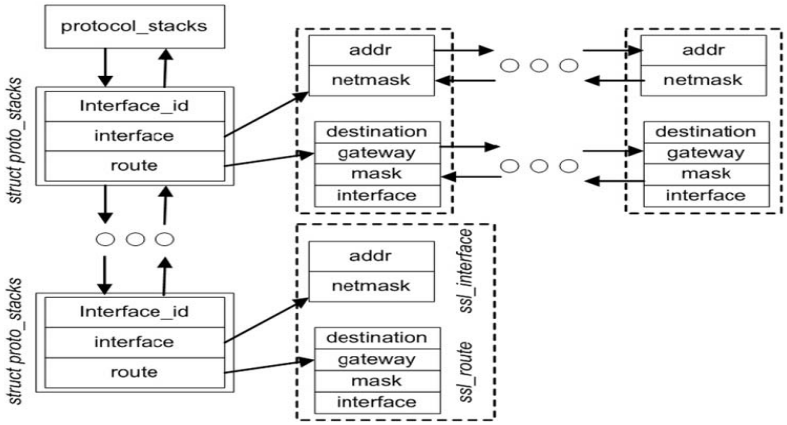
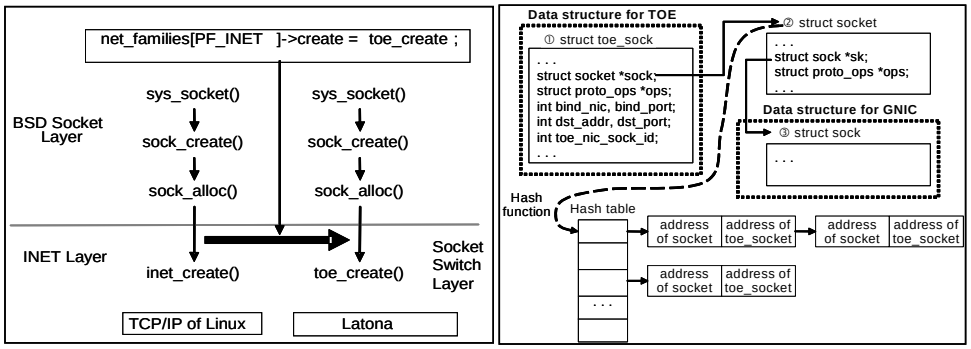


Fig. 2. The data structure of *proto_stacks.*

tional INET and TCP/IP protocol stack in a Linux server. It receives socket level commands from applications and delivers them to the LATONA hardware through the doorbell interface. The LATONA hardware is a kind of TOE which offloads the full TCP/IP stack and most parts of the socket processing [8-10].

The LATONA kernel is composed of Socket Switch Layer (SSL), Offloading Protocol Layer (OPL), and the LATONA driver as shown in Fig. 1. The socket level command generated by a network application is delivered to SSL through the BSD socket layer. Then, SSL determines whether it uses the LATONA hardware or a general NIC (GNIC) in serving this socket command according to the values of struct proto_stacks and the result of bind function. The struct proto_stacks shown in Fig. 2 has the information about the protocol stacks, such as routing information, network interface information, and interface_id field in order to identify a GNIC and a TOE device. If SSL deci-



(a) Socket creation. (b) Socket structure of SSL.

Fig. 3. The illustration of socket creation in LATONA kernel.

des to use the LATONA hardware, the command from the network application is delivered to the LATONA hardware through OPL and the LATONA driver. Otherwise, SSL handles this command by calling the INET layer of the legacy TCP/IP stack.

By locating SSL above OPL and the INET layer of the legacy stack, the LATONA kernel provides the binary compatibility with the BSD socket interface. It also supports both the legacy TCP/IP stack with a GNIC and the LATONA hardware. Thus, it is possible for traditional network applications to use the BSD socket interface without any modification and/or recompilation by using SSL that processes the standard BSD socket API.

For example, the socket creation of LATONA is similar to the traditional INET approach. The inet_create() function is replaced by toe_create(). The difference between inet_create() and toe_create() lies in the socket structure created. To support both GNIC and LATONA, toe_create() creates struct toe_sock which includes the conventional struct socket. Other connection information is also stored in struct toe_sock for later use. After creating a socket, struct toe_sock associated with a specific struct socket can be found in a hash table. This procedure is illustrated in Fig. 3. Similarly, all the other INET functions are supported by corresponding LATONA functions [9].

If SSL decides to use the LATONA hardware for data transmission, SSL calls OPL. Data transmission in OPL is performed by a true zero-copy mechanism through the DMA technique between the user memory and the LATONA hardware. For the true zero-copy, the user memory pages are pin down and physical addresses of these pages are passed to the LATONA hardware in the form of a scatter-gather list via the LATONA driver.

2.2 Hardware Architecture of LATONA

The LATONA hardware has a layered architecture as shown in Fig. 4. The host interface provides the communication channel between the kernel and hardware. It contains not only a PCI Express endpoint protocol engine, but also dual DMA engines and doorbells [10].

The DMA engines eliminate per-byte-cost from a host CPU by performing direct data movement between the payload buffer and the memory locations of the user address space. The doorbell interface provides simple and abstracted communications between the LATONA kernel and hardware.

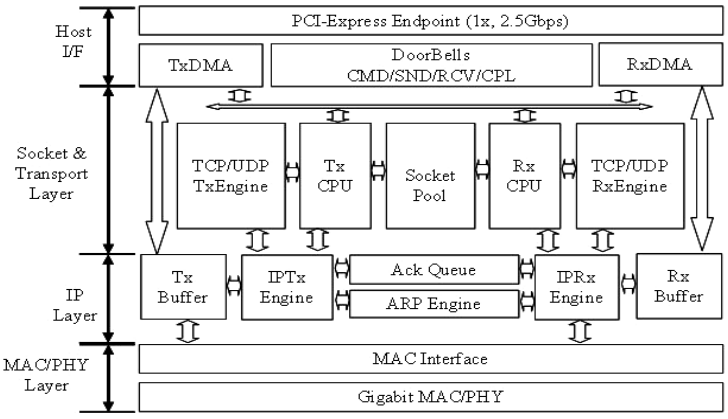


Fig. 4. The block diagram of LATONA hardware.

The transport layer processes the TCP and UDP protocols using both hardware and software. For software processing, two processors are embedded and each handles sending and receiving, respectively. The IP layer processes the IPv4 and ARP protocol, and the MAC/PHY layer sends and receives bit streams to/from Gigabit Ethernet.

The LATONA hardware employs a socket pool technique in order to offload the socket handling overhead from a host CPU. The socket pool can store 10K of socket entries and also contain the socket search logic in order to reduce the searching time for one entry. Each entry maintains TCP connection status, transaction IDs, and a scatter/gather list of each transaction for payload processing. Thus, the LATONA hardware can process socket level transactions without the interface with kernel software.

3. OFFLOADING SOCKET PROCESSING

Specifically, the object of this paper is to reduce the system overhead caused by the socket processing. LATONA has a hardware module to offload the socket handling overhead from a host CPU and specified data structures of the socket information in order to reduce the system overhead.

3.1 Hardware Module for Socket Processing

The hardware module shown in Fig. 5 is used where the TCP protocol is processed by a transmission processor and a reception processor. The transmission processor creates/destroys a socket, and the reception processor makes a request for search. The socket information is stored in an internal memory of the hardware device in the two processors. The hardware module includes the following components:

1. A TCP transmission processor which receives a request of creating/destroying a socket from a network program and executes the command.
2. A TCP reception processor which generates a search signal for a relevant socket ID when a new connect request packet is arrived.

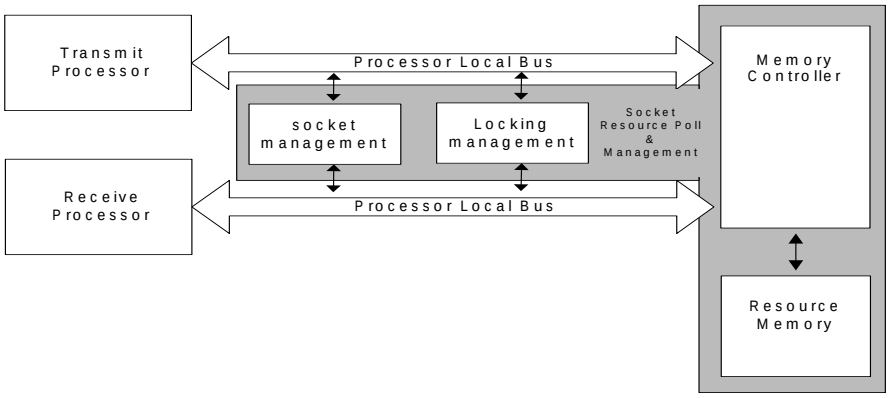


Fig. 5. The hardware module for offloading socket processing.

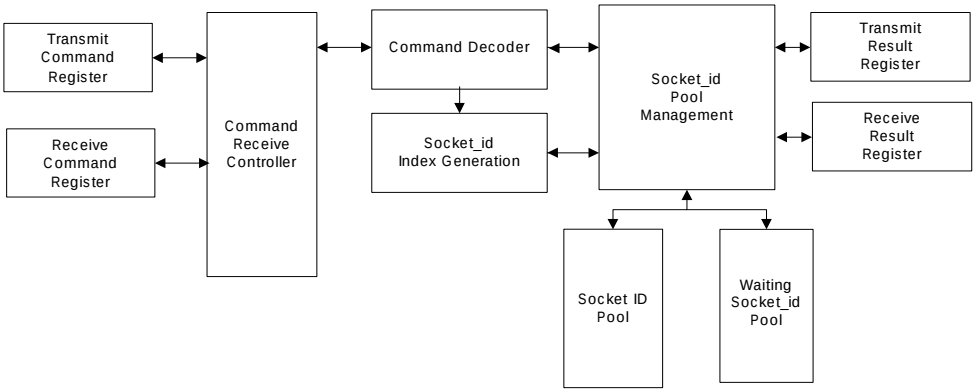


Fig. 6. The socket resource pool and management unit.

3. A socket management unit which creates/destroys a socket ID with a command from the TCP transmission processor and searches a socket ID with a command from the TCP reception processor.
4. A memory unit which stores the socket information under the control of the TCP transmission processor and provides the TCP reception processor with the socket information.
5. A locking management unit which controls concurrent accesses of the TCP transmission processor and the TCP reception processor to the same socket.

Furthermore, the memory unit includes a resource memory for storing the socket information and a memory controller for controlling the resource memory under the control of the TCP transmission processor or the TCP reception processor.

Fig. 6 describes a detailed block diagram of the socket resource pool and management unit shown in Fig. 5. The socket resource pool and management unit includes the following components:

1. A transmission command register for receiving a predetermined command from the

TCP transmission processor.

2. A reception command register for receiving a predetermined command from the TCP reception processor.
3. A command judging unit for checking commands in the transmit and the receive command register, and judging the contents of the commands.
4. A socket ID index generator for generating an index for searching a socket ID.
5. A socket ID pool for storing information of a socket ID, a destination IP address, a source port, and a destination port.
6. A wait socket ID pool for storing a socket ID of a relevant socket and a source port where a wait request is sent from an external system.
7. A socket ID pool management unit for controlling the socket ID pool or the wait socket ID pool with the command judged by the command judging unit.
8. A transmit and receive result register for storing the result of the command.

Fig. 7 shows a format of transmit/receive command and result register.

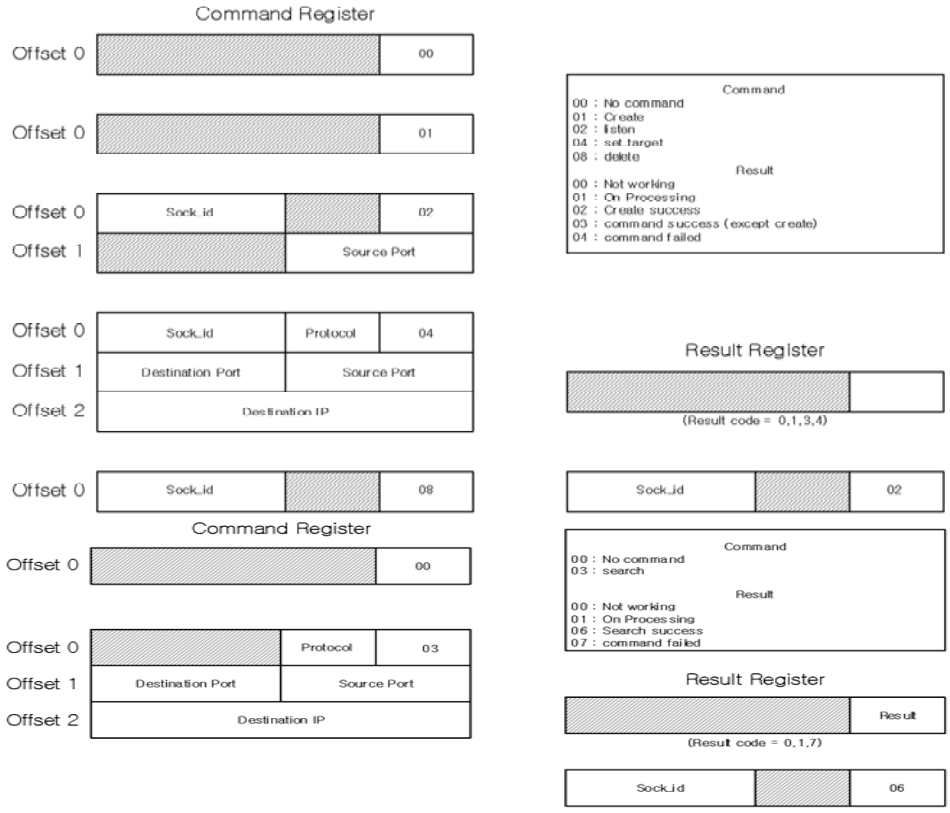


Fig. 7. Transmit/receive command and result register.

3.2 Data Structure for Socket Processing

The socket data structures of the Linux kernel have many elements in order to manage the protocol function, store the socket information, and transmit the network packet. However, the socket data structure for LATONA has to be limited to the minimum size because of the constraint of the hardware resource. Thus, we extract the necessary socket information for processing TCP/IP from struct sock and struct tcp_opt existed in Linux kernel [11]. Table 1 shows the dedicated socket data structure based on struct sock and struct tcp_opt for LATONA.

Table 1. Data structure for socket processing.

Size (bit)	Variable	Size (bit)	Variable	Size (bit)	Variable	Size (bit)	Variable	Size (bit)	Variable
32	socket_id	32	head_sgmt_ addr	8	urginline	8	tot_sgmt_no	8	dup_acks
16	src_port	32	recover	32	current_RTT	8	err	8	IP_RECVIF
8	sack_ok	32	keepalive_time	32	next_accept_sock	3	acked_byte_no	32	next_byte_no
16	sock_type	8	keepopen	16	parent_sock_id	8	size_option	32	snd_ssthresh
16	snd_mss	8	local_route	8	IP_TTL	16	urg_data	32	sened_sgmt_size
8	syn_retry	8	close_cmd_id	8	comp_sdb_id	32	bind_src_addr	32	tail_sgmt_addr
8	protocol	8	ack_pending	8	IP_RECVDSTADR	8	keepalive_probe	32	syn_rexmit_count
16	ack_backlog	32	err_soft	8	TCP_NODELAY	8	sock_state	8	probes_out
16	perv_ack_length	32	timestamp	32	before_fr_cwnd	8	rcv_wscale	8	debug
8	IP_TOS	32	urg_seq	32	connecthost_rwnd	32	keepalive_intvl	8	conn_cmd_id
8	ip_options_ length	32	dest_addr	32	sending_sgmt_addr	8	linger	8	use_sgmt_no
8	cong_ctrl_state	16	dest_port	32	sack_pipe	16	max_backlog	16	timeout
8	TCP_MAXSEG	8	tstamp_ok	16	prior_ssthresh	32	last_accept_sock	32	adv_window_size
32	unacked_byte_ no	8	snd_wscale	8	reuse	8	IP_HDRINCL	32	TCP_OPTION[10]
32	snd_cwnd	16	rcv_mss	8	broadcast	32	IP_OPTIONS[10]	32	recv_sgmt_addr

3.3 Socket Processing of LATONA

The major socket operations can be divided into four operations. A first operation is conducted when the TCP transmission processor makes a request for creating a socket in response to a request from a network program. A second operation is carried out when a socket destroy command is received from the network application. A third operation is executed when a wait of a socket is requested from a server network program. Finally, a fourth operation is performed when the TCP reception processor makes a request for searching a socket.

3.3.1 Creating and destroying socket

When a socket creation command is received from the TCP transmission processor, the socket management unit creates a new socket ID with reference to the stored socket IDs, and makes a socket ID creation success mark.

To be more specific, the TCP transmission processor receives a request for creation of a socket from a network program. Upon receipt of the request, the TCP transmission processor issues a command to a command field of the transmission command register by using code ‘01’, as shown in Fig. 7. A protocol field is classified into code ‘00000110’

indicating a connection-oriented protocol TCP and code ‘00010001’ indicating a non-connection-oriented protocol UDP. The command decoder interprets the received command and makes a request for creation of a new socket ID to the socket ID pool management unit if the command is for creating a socket. The socket ID pool management unit manages the socket ID pool where information including a socket ID, a destination IP address, a source port, and a destination port are stored. That is, it creates a socket ID with reference to the state of the socket ID pool. The created socket ID is stored in a socket ID field of the transmission result register. Then, the TCP transmission processor checks the result register, calculates an address to be stored in the resource memory of the memory unit on the basis of the relevant socket ID, and stores the socket information in a corresponding region.

When a socket destroy command is received from the TCP transmission processor, the socket management unit destroys the relevant socket ID from the stored socket IDs and then makes a success mark.

More specifically, when a request for socket destroy is received from a network program, the TCP transmission processor issues a command to a command field of the transmit command register by using ‘08’, as shown in Fig. 7. Then, the TCP transmission processor records the ID of the relevant socket in a socket ID field. Subsequently, the command decoder interprets the relevant command, and makes a request for destroying the relevant socket ID to the socket ID pool management unit. The socket ID pool management unit marks code ‘03’ indicating a command success in the transmission result register. The TCP transmission processor checks whether the command has succeeded or failed by checking the transmission result register.

3.3.2 Waiting socket

When the TCP transmission processor waits for a connection request from an external system, the TCP transmission processor sends a wait command to the socket management unit, and the socket management unit stores a relevant socket ID and a source port.

When a socket wait command is received from a network server program, the TCP transmission processor issues a command to the command field of the transmission command register by using code ‘02’, as shown in Fig. 7. The TCP transmission processor records the ID of the relevant socket in a socket ID field and the port information on the server program in a source port field.

Subsequently, the command decoder interprets the relevant command, and makes a request for wait of the relevant socket to the socket ID management unit. The socket ID management unit manages the wait socket ID pool where the information on a socket ID and a source port is stored. That is, it stores the socket ID and the source port in the wait socket ID pool, and marks code ‘03’ indicating a command success in the result field of the transmission result register. The TCP transmission processor checks whether the command has succeeded or failed by checking the transmission result register.

In the socket wait stage, the TCP reception processor sends the received information to the TCP transmission processor when a packet of connection request to the wait socket is received from a client system on a network. Then, the TCP transmission processor updates the socket information on the basis of the related information, and transmits a packet for connection establishment to the client system.

3.3.3 Searching socket

When a new packet is received from a client system, the TCP reception processor transmits a socket ID search command to the socket management unit. Then, the socket management unit creates an index for the search of a socket ID, searches a relevant socket ID by using the index, and then stores a success mark in the result register.

Specifically, when the TCP reception processor has received a new packet from a system connected to a network, it has to search the ID of the relevant socket. First, the TCP reception processor issues a command to the command field of the reception command register by using code '03'.

Then, the command decoder interprets the relevant command, and transmits the corresponding information to the socket ID index generator for generating an index for search. When a packet is received, the information for searching a corresponding socket ID includes a destination IP address, a source port, and a destination port. A general hashing table technique can be used for generating an index for search.

The index created by the socket ID index generator is transmitted to the socket ID pool management unit, and the socket ID pool management unit searches a socket ID in the socket ID pool on the basis of the index and the information transmitted from the command decoder. The socket ID pool management unit stores the searched socket ID in a socket ID field of the receive result register. Then, the TCP reception processor calculates an address to be read from the resource memory on the basis of the socket ID, and takes the socket information from the resource memory.

4. EXPERIMENTAL RESULTS

The LATONA has been implemented using FPGA with dual PPC405s operating at 250MHz. Also, we have developed 0.13μm CMOS ASIC-based LATONA which has dual ARM922Ts operating at 250MHz and consists of 3.4 million gates. Fig. 8 shows the LATONA card and ASIC.

In this paper, we focus on the effect of socket offloading which can be derived from the LATONA architecture. We first compared the kernel execution times of socket commands between the legacy Linux kernel and the LATONA kernel. Then, we compared the CPU overhead and bandwidth between GNIC and LATONA when network packets were transmitted to a client. The experimental system was a PC having a Pentium 2GHz

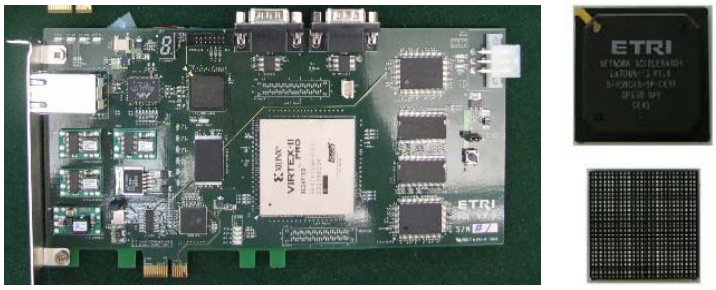


Fig. 8. The LATONA card and ASIC.

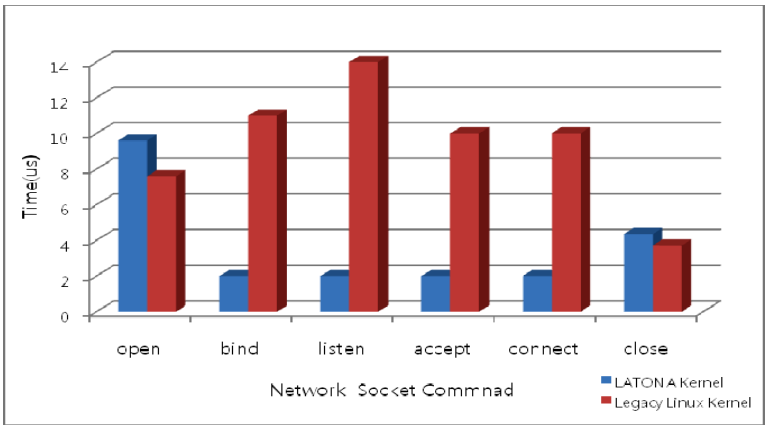


Fig. 9. The comparison of kernel execution time.

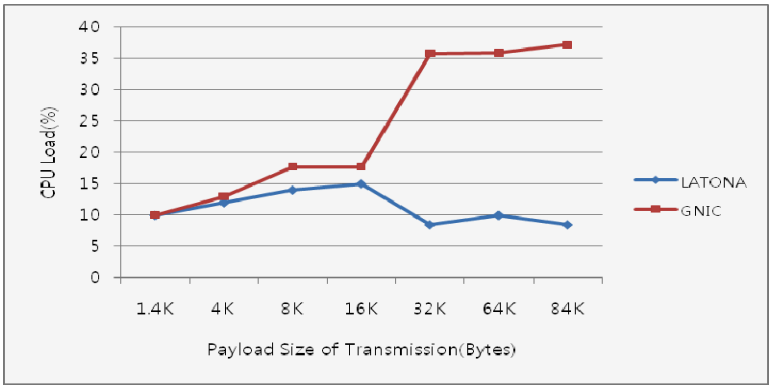


Fig. 10. The comparison of CPU overhead.

CPU, 512Mbytes of main memory. The operating system was Linux kernel 2.6.9. And, the GNIC compared was National Semiconductor's DP83820 Gigabit Ethernet card.

Fig. 9 shows the comparison of the kernel execution times of the network socket commands between GNIC and LATONA. The LATONA kernel execution times of socket creation and close commands were 1.3μs and 0.2μs longer than those of the legacy Linux kernel, respectively. This overhead is due to the fact that the LATONA kernel module maintains struct toe_sock in order to support both the LATONA and the GNIC. On the contrary, the legacy Linux kernel does not need to maintain this structure.

For other network commands, the LATONA kernel showed superior performance than the legacy Linux kernel. This is due to offloading the processing overhead (below the INET layer including socket processing and TCP/IP protocol processing) in LATONA.

Fig. 10 shows the comparison of the CPU overhead between GNIC and LATONA with various sizes of the transmission payload from 1.4Kbytes to 84Kbytes. LATONA can improve the CPU utilization of GNIC by a factor of four (with 84Kbytes). When the transmission data size increased from 16Kbytes to 32Kbytes, the legacy kernel consumed more CPU power to move data from the user memory to the payload buffer due to the limit

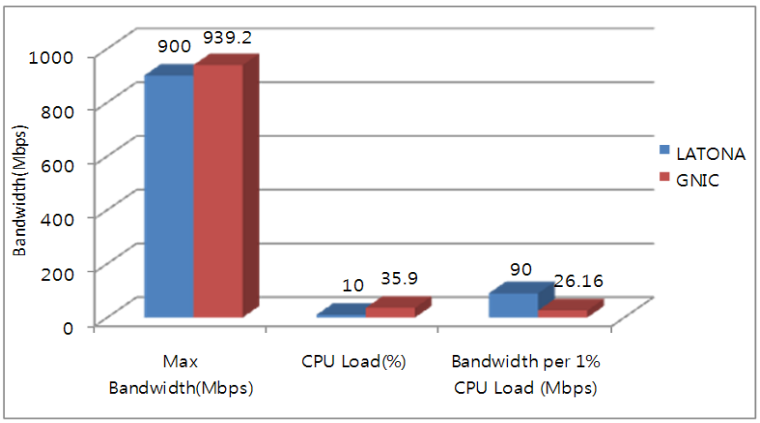


Fig. 11. The comparison of network bandwidth.

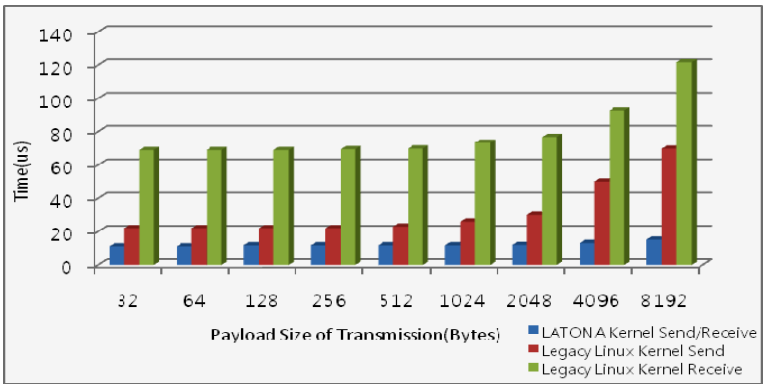


Fig. 12. The comparison of kernel execution time.

of the socket buffer size. Thus, the CPU overhead of GNIC jumped up sharply at 32Kbytes. On the other hand, the CPU overhead of LATONA was not increased, but decreased. This is because LATONA uses the DMA engine in order to move large data and the number of DMA commands requested by the host CPU is decreased. The reason of decreasing the number of DMA commands is that LATONA can transmit many network packets with one DMA command due to its large data size.

Fig. 11 shows the comparison of the maximum bandwidth between GNIC and LATONA. The maximum bandwidth of LATONA is less than that of GNIC, whereas the bandwidth per 1% CPU load of LATONA is greater than that of GNIC. Thus, LATONA can save the CPU cost for increasing the network speed.

Then, in order to measure the per-packet-cost accurately, we used 32bytes of send() and recv() commands. If the experimental data size is increased, the legacy Linux kernel should spend more time in order to move data from host CPU to GNIC. For small sized packet transfers, the data transfer time of GNIC in the legacy TCP/IP stack is negligible.

Fig. 12 shows the comparison of the kernel execution time of a message processing between the legacy Linux kernel and the LATONA kernel. In case of 32bytes with the

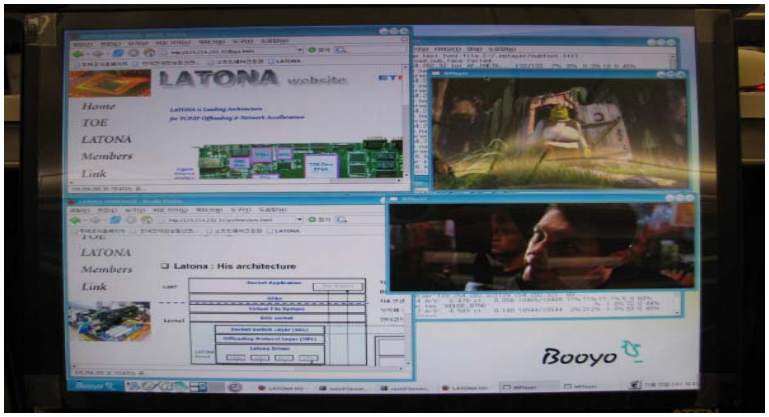


Fig. 13. The apache web server application.

legacy Linux kernel, INET, TCP/IP, and device driver spent 22μs and 69μs for send() and recv(), respectively. However, the LATONA kernel spent only 11μs for both send() and recv(). The kernel execution time of the legacy Linux kernel increases proportionally to the message size. On the contrary, the LATONA kernel spends fixed amount of time regardless of the message size. Consequently, LATONA can handle the terminal-initiated traffics required for the ubiquitous environment more efficiently, and thus utilize the saved time for processing another application programs.

Finally, Fig. 13 shows the execution result of the Apache web server application on LATONA in order to show the binary compatibility with the standard socket interface.

5. CONCLUSION

In this paper, we have presented the hardware and software of LATONA for offloading the socket processing. Based on the experimental results, LATONA could improve the network performance of a GNIC by a factor of four. Furthermore, it supports the binary compatibility with the standard socket interface such that the existing network programs can be used without modification and/or recompilation.

We believe LATONA accelerating the network performance can be applied to ubiquitous services where better network performance is required.

REFERENCES

1. M. Matsumoto and T. Itho, "Study of server processing load evaluations in ubiquitous communication environments," in *Proceedings of International Symposium on Applications and Internet Workshop*, 2006, pp. 122-125.
2. D. Clark, *et al.*, "An analysis of TCP processing overhead," *IEEE Communications Magazine*, Vol. 27, 1989, pp. 23-29.
3. A. Foong, *et al.*, "TCP performance revisited," in *Proceedings of International Symposium on Performance Analysis of Systems and Software*, 2003, pp. 70-79.

4. J. Mogul, "TCP offload is a dumb idea whose time has come," in *Proceedings of the 9th International Workshop on Hot Topics in Operating Systems*, Vol. 5, 2003, pp. 25-30.
5. G. Regnier, *et al.*, "TCP onloading for data center servers," *IEEE Computer*, Vol. 37, 2004, pp. 46-56.
6. E. Yeh, H. Chao, V. Mannem, J. Gervais, and B. Booth, "Introduction to TCP/IP offload engine (TOE)," *10 Gigabit Ethernet Alliance*, 2002.
7. A. Earls, "Integrating TCP offload engines (TOEs) – A look at vendors and trends," http://www.ajilon.ca/Sites/ajilon/multimedias/IntegratingTCPOffloadEngines0.pdf, 2002.
8. K. Park, S. Oh, S. Kim, and Y. Chung, "A network I/O architecture for terminal-initiated traffics in an ubiquitous service server," *Lecture Notes in Computer Science*, Vol. 5200, 2008, pp. 161-170.
9. S. Oh, *et al.*, "An effective linux kernel module supporting TCP/IP offload engine on grid," in *Proceedings of International Conference on Grid and Cooperative Computing*, 2006, pp. 228-235.
10. Y. Kim, *et al.*, "LATONA: network I/O acceleration architecture," in *Proceedings of International SoC Design Conference*, 2007, pp. 42-45.
11. T. Herbert, *The Linux TCP/IP Stack*: *Networking for Embedded Systems*, Charles River Media, Hingham, Massachusetts, 2004.

Sunwook Kim received the B.S. degree from Chungbuk National University, Korea, the M.S. degree from Hanyang University, Korea in 1996 and 2001 respectively, all in Computer Science. And he is a Ph.D. candidate in Korea University. He joined Electronics and Telecommunications Research Institute (ETRI) in Daejeon, Korea in 2001 and he is working as a senior research staff. He developed a linux device driver of InfiniBand HCA (Host Channel Adapter) and TOE (TCP Offloading Engine). His research interests include network acceleration for 10Gb, I/O virtualization and desktop virtualization.

Seongwoon Kim received the B.S. degree from Pukyong Natinal University, Korea, the M.S. degree from Chungnam National University, Korea in 1987 and 1998 respectively. He received his Ph.D. degree from Chungnam National University, Korea in 2006. He joined Electronics and Telecommunications Research Institute (ETRI) in Daejeon, Korea in 1989 and he is working as a chief of team. His research interests include network acceleration for 10Gb, I/O virtualization and power management.

Kyoung Park received the B.E. and M.E. degrees in Computer Engineering from ChonBuk National University, Korea in 1991 and 1993, respectively. He joined Electronics and Telecommunications Research Institute (ETRI) in Daejeon, Korea in 1993 and he is working as a chief of team. He developed main memory board of a SMP system called TICOM-III, router switch of a high performance parallel system called SPAX and InfiniBand HCA. His research interests is computer architecture with a focus on multi-processor, memory hierarchy, advanced I/O architecture for next generation computing.

Yongwha Chung received his B.S. and M.S. degrees from Hanyang University, Korea in 1984 and 1986, respectively. He received his Ph.D. degree from the University of Southern California, U.S.A. in 1997. He joined ETRI in 1986 and he was working for developing high-performance computing systems. Since 2003, he has been a Professor at Korea University. His research interests include parallel architecture/algorithm for multimedia and security applications.