고성능 Hybrid TCP/IP Offload Engine 구현을 위한 TCP/IP 프로토콜 분석

(Analysis of TCP/IP Protocol for Implementing a High-Performance Hybrid TCP/IP Offload Engine)

장 한 국 † 오 수 철 †† 정 상 화 ††† 김 동 규 †††
(Hankook Jang) (Soo-Cheol Oh) (Sang-Hwa Chung) (Dong Kyue Kim)

요 약 기존의 컴퓨터 시스템에서는 인터넷의 대표적인 프로토콜인 TCP/IP가 호스트 CPU에서 처리되는데, 이는 호스트 CPU에 많은 부하(load)를 발생시켜 전체 시스템의 성능을 저하시키는 문제를 야기한다. 최근 이러한 문제점을 해결하는 방안으로서 네트워크 어댑터에서 TCP/IP를 처리하는 TCP/IP Offload Engine(TOE)에 대한 연구가 활발히 진행되고 있다. 본 논문에서는 Linux 운영체제에 내장된 TCP/IP 프로토콜 스택의 구조를 분석하고, 통신을 수행할 때 프로토콜 스택의 각 함수에서 소모되는 시간을 측정하여 호스트 CPU에 부하를 발생시키는 주 요인을 분석하였다. 이러한 분석을 바탕으로 CPU에 많은 부하를 발생시키는 기능은 하드웨어로 구현하고 나머지 기능들은 소프트웨어로 구현하는 Hybrid TOE 구조를 제안한다.

키워드 : TOE, TCP/IP Offload Engine, TCP/IP, 리눅스, 프로토콜 스택

Abstract TCP/IP, the most popular communication protocol, is processed on a host CPU in traditional computer systems and this imposes enormous loads on the host CPU. Recently TCP/IP Offload Engine (TOE) technology, which processes TCP/IP on a network adapter instead of the host CPU, becomes an important way to solve the problem. In this paper we analysed the structure of a TCP/IP protocol stack in the Linux operating system and important factors, which cause a lot of loads on the host CPU, by measuring the time spent on processing each function in the protocol stack. Based on these analyses, we propose a Hybrid TOE architecture, in which functions imposing much loads on the host CPU are implemented using hardware and other functions are implemented using software.

Key words : TOE, TCP/IP Offload Engine, TCP/IP, Linux, Protocol stack

1. 서 론

인터넷을 비롯한 많은 분야에서 널리 사용되고 있는 Ethernet 기술은 이미 1 Gbps(Gigabit per second) 대역폭의 Gigabit Ethernet을 넘어 10 Gigabit Ethernet 기술의 표준화가 이루어졌다. Ethernet 상에서 사용되는 대표적인 통신 프로토콜인 TCP/IP는 일반적으로 호스트 CPU에서 처리되는데, 이는 호스트 CPU에 프로토콜을 처리하는 부하를 유발하여 전체 시스템의 성능을 떨어트린다. 특히 네트워크의 물리적인 성능이 발전함에 따라 TCP/IP를 사용한 통신에서 프로토콜 처리의 부하가 더욱 커지고 있다[1].

이러한 문제를 해결하는 방안으로서 TCP/IP 프로토콜의 처리를 호스트 CPU가 아닌 네트워크 어댑터에서 전담하는 TOE(TCP/IP Offload Engine) 기술에 대한 연구가 진행되고 있다. 네트워크 어댑터 상에서 TCP/IP를 처리하게 되면 호스트 CPU에 가해지는 부하가 줄어들고, 호스트 CPU가 프로토콜 처리 이외의 실질적인 작업에 전념함으로써 전체 시스템의 성능이 향상되는 효과를 얻을 수 있다. 또한 호스트 CPU에서 실행되는 작업량이 늘어나더라도 통신의 성능을 계속 유지할 수 있게 된다. TOE 구현에 대한 연구는 현재까지 크게 두

· 이 논문은 한국학술진흥재단의 지원에 의하여 연구되었음 (지방연구중심대학 육성지원 사업, 차세대물류IT기술연구사업단)
† 비 회 원 : 부산대학교 컴퓨터공학과
hkjang@pusan.ac.kr
†† 비 회 원 : 부산대학교 BK21 사업단 교수
osc@pusan.ac.kr
††† 종신회원 : 부산대학교 컴퓨터공학과 교수
shchung@pusan.ac.kr
dkkim1@pusan.ac.kr
논문접수 : 2004년 8월 29일
심사완료 : 2005년 1월 19일

가지 방향으로 진행되어 왔다. 첫 번째 방법은 내장형 프로세서(embedded processor)를 네트워크 어댑터에 장착하여 소프트웨어로 TCP/IP를 처리하는 것으로, 구현이 쉽고 펌웨어(firmware) 업그레이드만으로 기능 개선이 가능하다는 장점을 가진다. 그러나, 네트워크 어댑터에 장착 가능한 내장형 프로세서의 성능이 호스트 CPU의 성능과 많은 차이가 있으므로 통신 성능이 낮다는 단점을 가진다. 두 번째 방법은 TCP/IP를 처리하는 전용 ASIC을 개발하고 이를 네트워크 카드에 탑재하는 하드웨어 방식으로, 통신의 성능을 고려하여 전용 ASIC을 최적화함으로써 우수한 성능을 얻을 수 있다는 장점이 있다. 그러나, 전용 ASIC을 개발해야 하므로 개발 시간 및 비용이 많이 든다는 단점이 있다.

본 논문에서는 Linux 운영체제에 내장된 TCP/IP 프로토콜 스택의 구조를 분석하고, 통신을 수행할 때 프로토콜 스택의 각 함수에서 소모되는 시간을 측정하여 호스트 CPU에 부하를 발생시는 주 요인을 분석하였다. 분석 결과에 따르면, TCP/IP의 전체 기능들 중에서 일부 주요 기능들이 전체 통신 시간의 70~90%를 차지하였다. 따라서, 이러한 분석을 바탕으로 하여 하드웨어 구현과 소프트웨어 구현의 장점을 조합한 Hybrid TOE 구조를 제안한다. Hybrid TOE 구조는 TCP/IP 프로토콜을 사용한 송수신에서 시간을 많이 차지하는 기능들을 하드웨어로 구현함으로써 통신을 보장하고, 통신의 성능에 큰 영향을 끼치지 않는 기능들은 소프트웨어로 구현함으로써 개발에 소요되는 비용을 최소화하고자 한다.

본 논문의 구성은 다음과 같다. 2장에서 관련 연구를 소개하고, 3장에서 Linux TCP/IP 프로토콜 스택의 구조를 분석한다. 4장에서는 실험을 통해 TCP/IP 프로토콜 스택 내부에서 발생하는 오버헤드를 분석하고, 이러한 분석을 바탕으로 효율적인 TOE 구현 방안을 제안한다. 마지막으로 5장에서는 결론 및 향후 연구 과제를 제시한다.

2. 관련연구

TCP/IP 프로토콜 스택에서 발생하는 오버헤드를 분석한 결과에 따르면, TCP/IP의 오버헤드는 data-touching operation에서 발생하는 오버헤드와 nondata-touching operation에서 발생하는 오버헤드로 분류된다[2]. Data-touching operation은 데이타 복사 및 체크섬 계산과 같이 데이타 전체에 대해 접근하는 operation을 말하며, nondata-touching operation은 그 이외의 프로토콜 처리 operation을 가리킨다. Data-touching operation에서 발생하는 오버헤드를 줄이는 방법으로는 프로토콜 스택의 각 계층이 하나의 통신 버퍼를 공유함으로써 데이타의 불필요한 복사를 방지하는 single-copy 기법[3] 등이 제시되었다. Nondata-touching operation에서 발생하는 오버헤드를 줄이는 방법으로는 인터럽트 처리의 오버헤드를 줄이는 Interrupt Coalescing[4] 기법 등이 연구되었다. Interrupt Coalescing 기법은 매 패킷이 송·수신될 때마다 인터럽트를 발생시키는 것이 아니라 다수의 송·수신 인터럽트를 하나로 통합하여 발생시키고, 통합된 인터럽트에 의해 호출된 인터럽트 핸들러가 다수의 패킷을 한꺼번에 처리하는 방법이다.

TCP/IP를 최적화하려는 연구들에도 불구하고, 네트워크의 물리적인 속도가 기가비트급을 넘으면 호스트 CPU에서 TCP/IP를 처리하는 데 발생하는 부하를 근본적으로 해결하는 데 한계에 봉착한다[1]. 이러한 문제를 해결하기 위해 최근에는 호스트 CPU가 아닌 네트워크 어댑터에서 자체적으로 TCP/IP를 처리하는 TOE 기술이 실용화되고 있으며, 구현 방법으로는 네트워크 어댑터 상에 범용의 내장형 프로세서를 탑재하여 TCP/IP를 처리하는 소프트웨어 구현 방안과 ASIC 같은 전용 하드웨어로 TCP/IP를 처리하는 하드웨어 구현 방안이 제시되고 있다.

소프트웨어 기반 TOE 구현은 TCP/IP의 모든 기능을 소프트웨어로 처리하므로 구현이 쉽다는 장점을 가진다. 또한 기능을 개선하거나 TCP/IP 이외의 프로토콜을 처리하는 기능을 추가하기가 용이하다. 소프트웨어 기반 TOE 구현의 사례로는 Intel사의 PRO/1000T IP Storage Adapter[5]가 있다. 이 제품은 기존의 Gigabit Ethernet 어댑터에 Intel 80200 StrongARM(200MHz) processor를 장착하였으며, TCP/IP와 iSCSI(SCSI over IP) 프로토콜[6]을 소프트웨어로 처리한다. 콜로라도 대학에서 이 어댑터를 사용하여 실험한 결과[7]에 따르면 단방향 대역폭이 최대 30MB/s를 넘지 못하고 있는데, 이는 TCP/IP를 사용하는 일반적인 Gigabit Ethernet 어댑터가 최대 70MB/s 정도의 단방향 대역폭을 보여주는 것과 비교했을 때 성능이 2배 이상 낮은 것이다. 이렇게 소프트웨어 기반 TOE 구현은 통신의 성능 측면에서 약점을 가지며, 네트워크의 속도가 빨라질수록 그 차이가 더욱 커진다.

이러한 소프트웨어 기반 TOE의 성능 문제는 하드웨어로 TOE를 구현함으로써 해결할 수 있다. 하드웨어 기반 TOE 구현의 사례로는 Alacritech사의 SLIC (Session-Layer Interface Control)[8], QLogic사의 ISP4010[9], Adaptec사의 NAC-7711[10] 등이 있으며, 통신 성능으로는 단방향 대역폭이 100MB/s(800Mbps)를 넘어 Gigabit Ethernet의 성능을 최대한 발휘하는 것으로 알려져 있다[11,12]. 그러나, 하드웨어 기반 TOE 구현은 ASIC 구현에 많은 시간과 비용이 소모되고, 구

현된 하드웨어에서 수정하거나 개선할 사항이 발생할 때마다 새로운 ASIC을 개발해야 한다는 단점을 가진다. 특히 가까운 미래에 IPv6[13] 기반의 TOE에 대한 요구가 증가할 것으로 예상되는데, 현재까지 개발된 IPv4 기반의 TOE 중에서 하드웨어 기반 TOE의 경우 이에 신속히 대응하기가 어렵다. 또한 RDMA(Remote Direct Memory Access)[14] 등과 같은 새로운 상위 프로토콜까지 네트워크 어댑터에서 처리하려는 요구가 발생할 때에도 이에 효과적으로 대응하기 어렵다.

3. Linux 프로토콜 스택 분석

TCP/IP 프로토콜은 BSD Unix에서 성공적으로 구현되었고, 이후 대부분의 운영체제에 기본적으로 구현되어 인터넷의 표준 프로토콜로 사용되어 왔다. 초기의 TCP/IP는 프로토콜의 각 계층을 지날 때마다 이전 계층에서 만들어진 패킷 전체를 먼저 복사한 후 헤더를 덧붙이는 과정을 반복하는 비효율적인 구조로 구현되었고, 이는 통신의 성능을 저하시키는 주요 요인이 되었다. 반면 Linux에서는 TCP/IP를 효율적으로 운용하기 위해 소켓 버퍼 구조체를 통해 불필요한 복사를 제거하는 등의 연구가 활발하게 진행되었고, 그 결과 운영체제들 중에서 비교적 우수한 네트워크 성능을 제공하기 때문에 본 논문에서 TCP/IP 프로토콜 스택의 분석을 위한 운영체제로 선택하였다.

3.1 Linux 프로토콜 스택의 구조

Linux의 TCP/IP 프로토콜 스택은 그림 1과 같은 구조를 가진다.

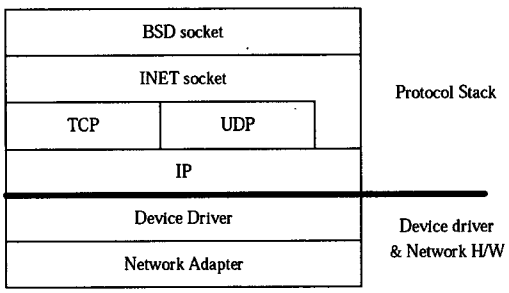

그림 1 Linux TCP/IP 프로토콜 스택의 구조

최상위 계층에는 BSD 소켓(socket)을 지원하는 BSD 소켓 계층(layer)이 자리잡고 있으며, 사용자가 UNIX 표준의 소켓 인터페이스를 통해 TCP/IP를 비롯한 다양한 통신 프로토콜을 사용할 수 있도록 지원한다. BSD 소켓 계층의 아래에는 INET 소켓 계층이 자리잡고 있으며, 실제 통신에 사용되는 프로토콜(TCP, UDP, raw IP)에 대한 abstraction을 제공한다. INET 계층의 하부에는 TCP/UDP 계층이 있고, 그 밑에 IP 계층이 위치한다. 커널의 프로토콜 스택 하부에는 네트워크 어댑터의 디바이스 드라이버가 자리잡고 있다.

Linux의 TCP/IP 프로토콜 스택에서는 대표적인 data-touching operation인 데이타 복사를 최소화하기 위해 소켓 버퍼 구조체[15]를 사용한다. 소켓 버퍼는 데이타 영역 및 프로토콜 스택의 각 계층에서 생성하는 헤더들을 모두 저장할 수 있는 헤더 영역으로 구성된다. 송신 과정에서는 TCP 계층에서 전송을 시작하는 초기에 소켓 버퍼를 생성하며, 전송할 데이타는 헤더 영역의 다음에 이어지는 데이타 영역으로 복사된다. 이후 각 계층의 함수들을 통과할 때에는 소켓 버퍼에 대한 포인터가 전달되며, 각 계층에서는 이 포인터를 통해 이전 계층에서 만들어진 헤더의 앞쪽에 자신이 만든 헤더만을 덧붙임으로써 불필요한 데이타 복사를 방지한다. 수신 과정에서는 디바이스 드라이버가 소켓 버퍼를 생성하여 수신된 패킷을 저장하고, 이후 각 계층에서는 자기 계층에 해당하는 헤더를 제거한 후 상위 계층으로 소켓 버퍼에 대한 포인터를 넘긴다.

3.2 Linux의 TCP/IP 송신 과정

Linux의 TCP/IP를 사용한 송신 과정에서 거치는 함수들의 경로는 그림 2와 같다.

사용자 프로그램에서 *write()* 또는 *send()* 함수를 사용하여 송신을 요청하면, BSD 소켓 계층에서 이들에 대응하는 *sys_write()* 함수 또는 *sys_send()* 함수가 시스템 콜(system call)을 통하여 호출된다. 이어서 송신할 소켓의 정보가 *sock_sendmsg()* 함수와 *inet_sendmsg()* 함수를 거쳐 TCP 계층의 *tcp_sendmsg()* 함수로 전달된다.

TCP 계층의 *tcp_sendmsg()* 함수에서 수행하는 작업들은 다음과 같다. 먼저 TCP 계층에서 생성할 패킷의 크기를 결정하기 위해 MSS(Maximum Segment Size)를 조사하는데, 이는 일반적으로 네트워크 어댑터의 MTU(Maximum Transfer Unit) 크기에 따라 결정된다. MSS가 결정되면 *tcp_alloc_pskb()* 함수를 통해 소켓 버퍼의 생성을 요구하며, 이를 위해 TCP 연결(connection)이 설정될 때 64KB씩 할당받은 송신 버퍼 영역에서 소켓 버퍼를 생성할 여유 공간이 있는지 확인한다. 만약 데이타의 크기가 사용 가능한 송신 버퍼의 크기보다 큰 경우에는 확보된 버퍼 용량만큼의 데이타를 먼저 처리한 후 송신 프로세스의 진행을 멈추고 송신 버퍼가 확보되기를 기다린다. 이를 위해 호출된 *wait_for_tcp_memory()* 함수는 sleep/wake-up 메커니즘 기반의 Linux 프로세스 스케줄러를 사용하여 현재의 송신 프로세스를 sleep 상태로 바꾼다. Sleep 상태로 들어간 송신 프로세스는 송신 버퍼에 충분한 메모리가 확

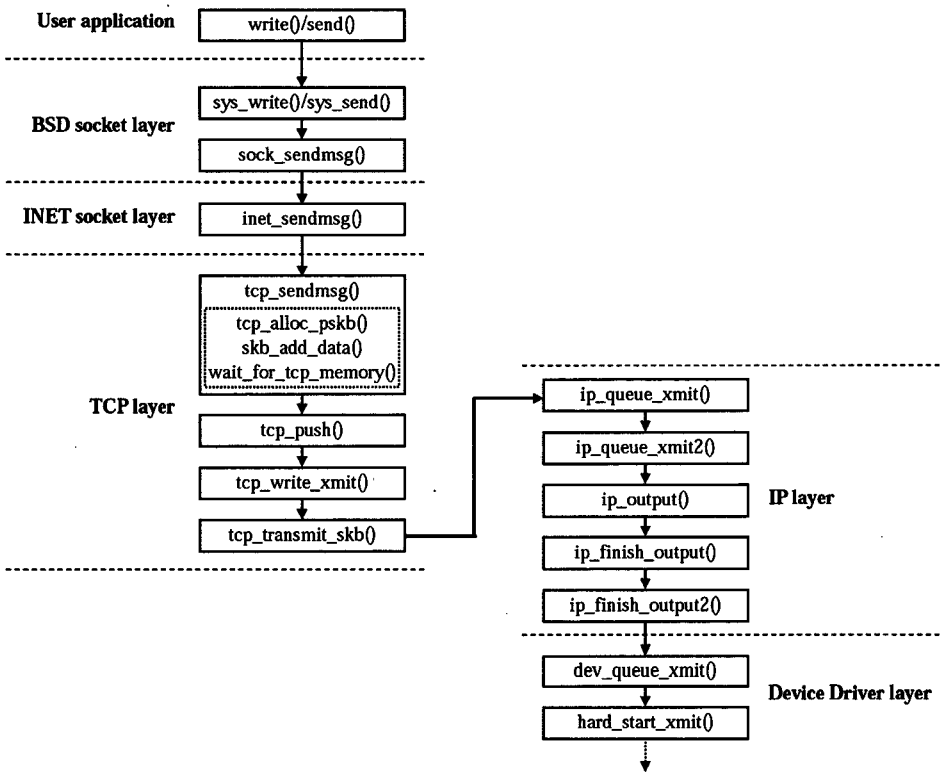

그림 2 Linux의 TCP/IP 송신 과정에서 거치는 함수 경로

보되면 수행을 재개한다. 이 때 송신할 데이타의 양이 증가할수록 송신 버퍼에서 필요한 공간을 확보하기가 어려워지므로 프로세스 스케줄러를 호출하는 회수도 더욱 늘어나게 되며, 따라서 프로세스들 사이의 작업 전환에 걸리는 시간이 누적되어 전체적으로 오버헤드가 증가할 것으로 예상된다.

tcp_alloc_pskb() 함수를 통해 소켓 버퍼가 만들어지면 *skb_add_data()* 함수에서 사용자 데이타를 새로 생성된 소켓 버퍼로 복사한다. 이는 data-touching operation에 속하며, 이 과정에서 발생하는 오버헤드는 데이타의 크기에 비례하여 증가하게 된다. 데이타가 복사된 소켓 버퍼는 TCP 계층과 IP 계층 사이에서 송신할 소켓 버퍼들을 관리하는 write queue에 등록된다. *tcp_sendmsg()* 함수의 작업이 끝나면 write queue에 등록된 소켓 버퍼들은 *tcp_push()* 함수, *tcp_write_xmit()* 함수, *tcp_transmit_skb()* 함수를 거쳐 처리된다. 이 과정에서 순서 번호(sequence number) 할당과 TCP 체크섬(checksum) 계산 등의 작업을 통해 TCP 헤더가 완성된다. TCP 헤더가 완성되면 TCP 계층의 마지막 함수인 *tcp_transmit_skb()* 함수는 IP 계층의 *ip_queue_xmit()* 함수를 호출하여 소켓 버퍼에 대한 포인터를 IP 계층으로 전달한다.

IP 계층의 시작점인 *ip_queue_xmit()* 함수에서는 TCP 계층에서 넘겨받은 소켓 버퍼에 목적지 IP 주소와 출발지 IP 주소 등의 IP 헤더 필드를 채운다. *ip_queue_xmit2()* 함수에서는 IP 헤더의 체크섬 계산 등이 수행되고, 이어 *ip_output()* 함수와 *ip_finish_output()* 함수를 거치면서 실제 송신에 사용될 네트워크 어댑터가 지정된다. *ip_finish_output2()* 함수는 디바이스 드라이버 계층의 *dev_queue_xmit()* 함수를 호출한다. IP 계층에서의 처리를 마친 소켓 버퍼는 디바이스 드라이버 계층의 *dev_queue_xmit()* 함수에 전달되어 네트워크 어댑터를 통해 상대 노드로 전송된다. IP 계층에서는 20 byte 크기의 IP 헤더에 대한 체크섬 계산 이외에는 데이타 패킷에 접근하는 작업이 없기 때문에 data-touching operation에 의한 오버헤드가 크지 않을 것으로 판단된다. 또한 흐름 제어나 혼잡 제어 등의 복잡한 작업이 거의 없기 때문에 nondata-touching operation에 의한 오버헤드도 TCP 계층에 비해 작을 것으로 예상된다.

3.3 Linux의 TCP/IP 수신 과정

Linux의 TCP/IP 수신 과정에서 거치는 함수들의 흐름은 그림 3과 같다.

수신 과정은 2개의 분리된 프로세스에 의해 수행되며, 데이타가 수신되기를 기다리는 수신대기 프로세스와 네트워크 어댑터에서 패킷이 수신되었을 때 수신 작업을 처리하는 수신처리 프로세스로 이루어진다.

데이타가 수신되기를 기다리는 수신대기 프로세스는 사용자 프로그램에서 *read()* 또는 *recv()* 함수의 호출을 통해 시작되고, 시스템 콜에 의해 BSD 소켓 계층의 *sys_read()* 함수나 *sys_recv()* 함수가 호출되면서 커널에 진입한다. 그리고, BSD 소켓 계층의 *sock_recvmsg()* 함수와 INET 계층의 *inet_recvmsg()* 함수를 거쳐 TCP 계층의 *tcp_recvmsg()* 함수에 도달한다. *tcp_recvmsg()* 함수는 현재 프로세스의 상태를 sleep 상태로 바꾸고 데이타의 수신을 기다린다. Sleep 상태로 들어간 수신대기 프로세스는 *sock_def_readable()* 함수가 수신처리 프로세스에 의해 호출될 때 깨어난다. 깨어난 수신대기 프로세스는 호출된 함수들을 역순으로 거치면서 수신 데이타를 처리하는 작업을 마무리한다.

수신처리 프로세스는 네트워크 어댑터가 패킷을 수신한 후 발생시키는 인터럽트에 의해 시작된다. 인터럽트가 발생하면 interrupt handler가 디바이스 드라이버의 reception function을 작동시켜 수신된 패킷을 커널로 가져온다. Reception function은 소켓 버퍼를 생성하여 수신된 패킷을 저장하고, *netif_rx()* 함수에 소켓 버퍼에 대한 포인터를 넘김으로써 상위의 IP 계층으로 수신된 패킷을 전달한다. IP 계층에서는 *ip_rcv()*, *ip_rcv_finish()* 함수를 거치면서 목적지 IP 주소, 패킷의 길이, IP 버전 번호, IP 헤더 체크섬 등을 검사하여 처리한 후 소켓 버퍼에서 IP 헤더를 제거한다. 이후 마지막으로 *ip_local_deliver()* 함수를 통하여 TCP 패킷만 남은 소켓 버퍼를 TCP 계층으로 전달한다. IP 계층에서는 수신 과정에서도 송신 과정과 마찬가지로 헤더 처리 이외의 특별한 작업이 거의 없으므로 오버헤드가 크지 않을 것으로 예상된다.

TCP 계층에서는 *tcp_v4_rcv()* 함수를 시작으로 *tcp_rcv_established()* 함수를 거치면서 TCP 패킷을 처리한다. *tcp_v4_rcv()* 함수에서는 수신된 TCP 패킷의 체크섬 검사, 순서 번호 검사 등을 수행하고, TCP 헤더에

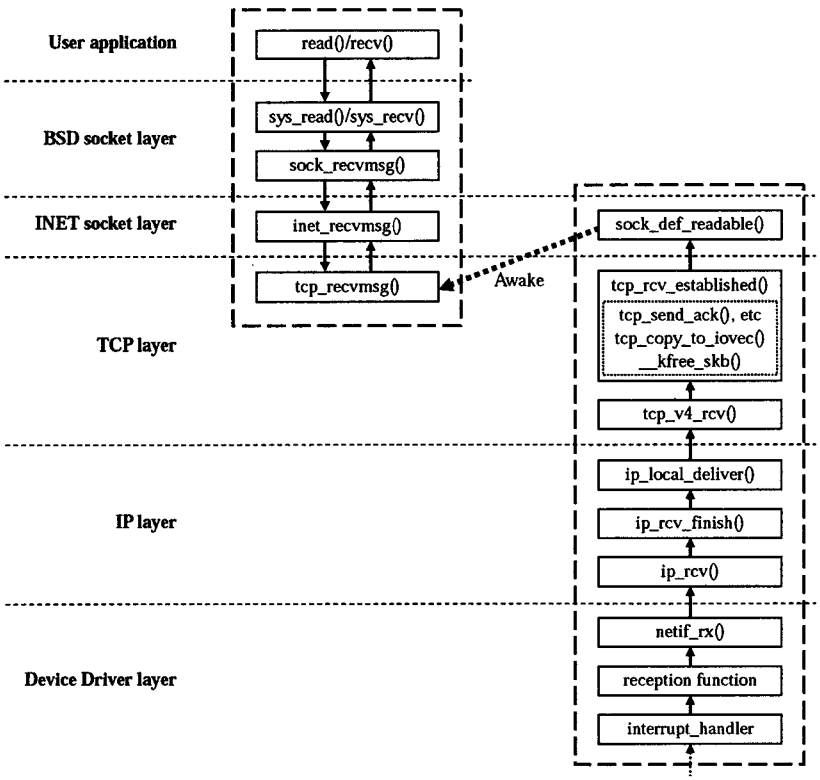

그림 3 Linux의 TCP/IP 수신 과정에서 거치는 함수 경로

서 source port와 destination port의 port number를 해석하여 소켓 버퍼를 전달할 INET 소켓을 찾는다. *tcp_rcv_established()* 함수에서는 ACK 패킷을 처리하는 과정, 수신 데이타를 INET 소켓 계층으로 복사하는 과정, 소켓 버퍼를 해제하는 과정 등의 주요 작업들을 처리한다. 이 기능들은 각각 *tcp_send_ack()* 함수를 포함한 ACK 처리 함수들, *tcp_copy_to_iovec()* 함수, *__kfree_skb()* 함수로 구현되어 있다. 이후 INET 소켓 계층의 *sock_def_readable()* 함수를 통해 sleep 상태의 수신대기 프로세스들을 깨운다. TCP 계층에서는 IP 계층과 달리 데이타 전체를 복사하는 data-touching operation이 포함되고 또한 ACK 패킷을 회신하는 과정과 소켓 버퍼를 해제하는 과정까지 포함하므로 IP 계층에 비해 상대적으로 큰 오버헤드를 가질 것으로 예상된다.

4. Linux 프로토콜 스택의 오버헤드 분석 및 Hybrid TOE 구조

본 장에서는 Linux의 TCP/IP 프로토콜 스택을 사용한 통신에서 주요 함수를 통과하는 데 걸리는 시간을 측정하고, 이를 분석하여 효율적인 TOE 구현 방안을 제안한다. 본 실험에 사용된 두 대의 컴퓨터 노드는 각각 Intel SE7500CBW2 메인보드 상에 한 개의 1.8GHz Intel Xeon CPU와 주 메모리로 512MB 용량의 registered DDR SDRAM을 장착하였다. 네트워크 어댑터는 Intel PRO/1000MT Gigabit Ethernet Server Adapter를 사용하였다. 운영체제로는 2.4.7-10 버전의 커널을 기반으로 하는 RedHat 7.2 Linux package를 채택하였고, 네트워크 어댑터의 디바이스 드라이버로는 Intel에서 제공하는 e1000-5.2.20 버전을 사용하였다. 2대의 컴퓨터 노드는 스위치를 거치지 않고 직접 연결하였다.

본 논문에서는 실험을 통해 TCP/IP를 사용한 송수신 과정의 각 함수에서 소요되는 시간을 측정하였다. 시간의 단위로는 CPU 클럭수를 사용하였고, 작은 데이타부터 큰 데이타까지 통신에서 소요되는 시간을 비교하기 위해 1.4 KB, 14 KB, 140 KB, 1 MB 크기의 데이타를 사용하였다.

4.1 TCP 계층과 IP 계층의 소요시간 비교

표 1에서는 송수신 과정에서 TCP 계층과 IP 계층에서 소요되는 시간과 비율을 제시하였다.

데이타의 크기가 1.4 KB~14 KB 사이일 경우에는 송신과 수신 양쪽에서 TCP 계층이 전체 TCP/IP 프로토콜 처리 시간의 81~90%를 소모하였다. 데이타가 144 KB~1 MB의 크기일 경우에는 TCP 계층의 비중이 더 늘어나 전체 처리 시간의 92~95%를 차지하였다. 이는 TCP 계층에서 통신의 신뢰성을 보장하기 위해 처리하는 작업들이 IP 계층보다 많고 복잡하기 때문이다. 따라서, 효율적인 TOE 구조를 개발하기 위해서는 TCP 계층에 대한 최적화가 필수적이다. 이를 위해 본 논문의 4.2절과 4.3절에서는 송신 과정과 수신 과정에서 TCP 계층 내부의 소요 시간을 분석하였다.

4.2 TCP 송신 과정의 소요시간 분석

표 2에서는 TCP 계층의 송신 과정에서 많은 시간을 소모하는 주요 함수들의 수행 시간과 함께, TCP 계층에서 송신을 처리하는 시간(TCP 송신 시간)에 대한 각 함수의 수행 시간의 비율을 제시하였다.

송신 과정에서 가장 많은 시간을 소모하는 함수는 <그림 2>의 *tcp_sendmsg()* 함수였으며, 특히 *tcp_sendmsg()* 함수 내부에서 소켓 버퍼를 생성하는 *tcp_alloc_pskb()* 함수, 사용자 영역의 데이타를 소켓 버퍼로 복사하는 *skb_add_data()* 함수, 그리고 TCP 송신버퍼 영역에서 소켓 버퍼를 위한 공간이 확보되기를 기다리는 *wait_for_tcp_memory()* 함수의 수행시간들을 합친 것이 전체 TCP 처리 시간의 85~95%를 차지하였다.

송신 과정에서 주요한 3개의 함수들이 TCP 송신 시간에서 차지하는 비중은 전송할 데이타의 크기에 따라 양상이 달라졌다. 전송할 데이타의 크기가 1.4 KB~14 KB 정도로 비교적 작은 경우에는 *tcp_alloc_pskb()* 함수의 수행시간이 TCP 처리 시간의 60% 이상을 차지하였고, 다음으로는 *skb_add_data()* 함수가 약 20%의 비중을 차지하였다. 그리고, 데이타의 크기가 작은 경우 소켓 버퍼를 위한 메모리가 충분히 확보되므로 *wait_for_tcp_memory()* 함수는 호출되지 않았다.

반면에 전송할 데이타의 크기가 140 KB~1 MB로

표 1 TCP 계층과 IP 계층에서 소요되는 시간과 비율

(단위 : 1,000 CPU clocks)

		1.4 KB		14 KB		140 KB		1 MB	
송신 과정	TCP	14	(81%)	166	(90%)	5,651	(95%)	44,411	(95%)
	IP	3	(19%)	19	(10%)	267	(5%)	2,100	(5%)
	Total	17	(100%)	185	(100%)	5,918	(100%)	46,511	(100%)
수신 과정	TCP	48	(81%)	214	(82%)	1,816	(92%)	15,304	(94%)
	IP	11	(19%)	49	(18%)	158	(8%)	1,004	(6%)
	Total	59	(100%)	263	(100%)	1,974	(100%)	16,308	(100%)

표 2 TCP 계층의 주요 송신 함수에서 소요되는 시간과 비율

(단위 : 1,000 CPU clocks)

함수 \ 데이터 사이즈	1.4 KB		14 KB		140 KB		1 MB	
tcp_alloc_pskb()	9	(64%)	105	(63%)	957	(16%)	7,291	(16%)
skb_add_data()	3	(21%)	46	(27%)	413	(7%)	2,525	(5%)
wait_for_tcp_memory()	0	(0%)	0	(0%)	4,092	(72%)	31,298	(70%)
TCP 송신 시간	14	(100%)	166	(100%)	5,651	(100%)	44,411	(100%)

비교적 클 경우에는 *tcp_alloc_pskb()* 함수와 *skb_add_data()* 함수에서 소요되는 시간에 비해 *wait_for_tcp_memory()* 함수에서 소요되는 시간이 급격히 증가하여 TCP 송신 시간에서 70% 이상의 비중을 차지하였다. 이는 3.2절에서 설명한 바와 같이 sleep/wake-up 메커니즘 기반의 프로세스 스케줄러를 사용하는 것이 가장 큰 요인이다. 즉, 소켓 버퍼용 메모리를 확보하지 못한 송신 프로세스가 sleep 상태로 들어갈 때 sleep 상태와 wake-up 상태로 전이하는 작업 전환 과정에서 많은 오버헤드가 발생하고, 또한 일단 sleep 상태로 들어간 프로세스는 다시 깨어날 때까지 많은 시간을 기다려야하기 때문이다.

4.3 TCP 수신 과정의 소요시간 분석

표 3에서는 표 2와 같이 TCP 계층의 수신 과정에서 많은 시간을 소모하는 주요 함수들의 수행 시간과 함께, TCP 계층에서 수신을 처리하는 시간(TCP 수신 시간)에 대한 각 함수의 수행 시간의 비율을 제시하였다.

수신 과정에서는 TCP 헤더 검사와 처리를 담당하는 *tcp_v4_rcv()* 함수와 *tcp_rcv_established()* 함수 내부에서 호출되는 3 종류의 함수들을 수행하는 데 가장 많은 시간을 소모하였으며, TCP 수신 시간에서 85% 이상의 비중을 차지하였다. *tcp_rcv_established()* 함수 내부에서 호출되는 3 종류의 함수는 데이타를 INET 소켓으로 복사하는 *tcp_copy_to_iovec()* 함수, *tcp_send_ack()* 함수를 포함한 ACK 처리 함수들, 그리고 처리가 끝난 소켓 버퍼를 해제하는 *__kfree_skb()* 함수이다.

수신 과정에서도 송신 과정과 마찬가지로 데이타의 크기에 따라 주요 함수들의 비중이 달라졌다. 1.4 KB의 데이타를 수신할 때에는 *tcp_copy_to_iovec()* 함수의 비중이 35%로 가장 컸고, 다음으로 ACK 처리 함수들과 *tcp_v4_rcv()* 함수의 비중은 22~27% 정도로 비슷하였다. 14 KB 크기의 데이타를 수신하는 경우 *tcp_copy_to_iovec()* 함수와 ACK 처리 함수들의 비중은 큰 변화가 없었지만, *__kfree_skb()* 함수의 비중은 높아졌고 *tcp_v4_rcv()* 함수의 비중이 낮아졌다. 수신 데이타의 크기가 140 KB~1 MB 정도로 더 커지면 *tcp_copy_to_iovec()* 함수와 *__kfree_skb()* 함수의 수행 시간이 차지하는 비율이 급격히 높아졌고, ACK 처리 함수들과 *tcp_v4_rcv()* 함수의 비중이 크게 떨어졌다.

데이타의 크기가 커질 때 *tcp_v4_rcv()* 함수와 ACK 처리 함수들의 비중은 크게 떨어지고 대신 *tcp_copy_to_iovec()* 함수의 비중이 높아지는 이유는 캐쉬 효과에 의해 크기가 작은 TCP 헤더와 ACK 패킷을 처리하는 오버헤드가 데이타 전체를 복사하는 오버헤드에 비해 작기 때문인 것으로 분석된다. 즉, TCP 패킷의 헤더 영역과 ACK 패킷은 크기가 작을 뿐만 아니라 패킷을 만드는 과정에서 빈번한 접근이 발생하여 캐쉬의 활용도가 높은 데 반해 데이타는 크기가 비교적 크고 사용 빈도도 낮아 캐쉬 효과가 거의 발생하지 않기 때문인 것으로 판단된다. 데이타의 크기가 커질 때 소켓 버퍼를 해제하는 *__kfree_skb()* 함수의 비중이 높아지는 이유는 이 함수에서 메모리를 커널에 반납하고 버퍼 리스트를 설정하는 등의 복잡한 처리 과정을 거치는 오버헤드가 데이타의 크기에 비례하여 증가하는 것이 원인으로 파악된다.

마지막으로 표 4에서는 4.2절과 4.3절의 실험 결과를 바탕으로 효율적인 TOE 구조를 개발하는 데 고려해야 할 사항들을 제시한다. 표 2에서 제시한 송신 과정의 주

표 3 TCP 계층의 주요 수신 함수에서 소요되는 시간과 비율

(단위 : 1,000 CPU clocks)

함수 \ 데이타 사이즈	1.4 KB		14 KB		140 KB		1 MB	
tcp_v4_rcv()	11	(22%)	12	(5%)	117	(6%)	720	(4%)
ACK functions	13	(27%)	54	(25%)	99	(5%)	294	(2%)
tcp_copy_to_iovec()	17	(35%)	72	(33%)	767	(42%)	7,413	(48%)
__kfree_skb()	5	(10%)	51	(23%)	720	(39%)	5,710	(37%)
TCP 수신 시간	48	(100%)	214	(100%)	1,816	(100%)	15,304	(100%)

표 4 TCP 계층에서 최적화해야 할 함수들

	함 수	설 명
송신 과정	① tcp_alloc_pskb()	TCP 송신버퍼 영역에 소켓 버퍼 생성
	② skb_add_data()	사용자 영역의 데이타를 소켓 버퍼로 복사
	③ wait_for_tcp_memory()	소켓 버퍼를 위한 공간이 확보될 때까지 대기
수신 과정	④ tcp_v4_rcv()	TCP 헤더 검사 및 처리
	⑤ ACK functions	ACK 패킷 생성 및 전송 처리
	⑥ tcp_copy_to_iovec()	소켓 버퍼의 수신 데이타를 사용자 영역으로 복사
	⑦ __kfree_skb()	소켓 버퍼 해제

요 함수 3개는 TCP/IP 송신 시간의 70% 이상을 차지하고, 표 3에서 제시한 수신 과정의 주요 함수 4개는 TCP/IP 수신 시간의 77% 이상을 차지하므로 이 함수들을 최적화하는 것이 TOE 구현에서 가장 중요한 요소로 판단된다.

4.4 Hybrid TOE 구조

표 4에 정리된 기능들은 TCP/IP를 사용한 통신 시간에서 70~90%를 차지하지만, TCP/IP를 구성하는 전체 기능들 중에서는 일부분에 불과하다. 본 절에서는 이러한 분석을 바탕으로 하여, 통신의 성능에 큰 영향을 끼치는 기능들을 하드웨어로 구현하고 통신의 성능에 영향이 적은 나머지 기능들은 소프트웨어로 운용하는 Hybrid TOE 구조를 제안한다. 본 논문에서 제안하는 Hybrid TOE 구조는 처리에 시간이 많이 걸리는 주요 기능들을 하드웨어로 구현함으로써 통신의 성능을 보장할 수 있고, 수행 시간의 비중이 높지 않은 기능들은 소프트웨어로 쉽게 구현할 수 있을 것으로 기대된다. Hybrid TOE의 내부 구조는 그림 4와 같다.

본 논문에서 제안하는 Hybrid TOE 구조에서는 표 4에서 제시한 기능들 중 ③을 제외한 나머지 6개의 기능들을 하드웨어로 구현하고, ③은 두 개의 내장형 프로세서 코어(core)를 사용하여 소프트웨어로 구현한다. Hybrid TOE 구조에서는 소켓 버퍼가 컴퓨터의 주메모리가 아닌 TOE 어댑터 상의 버퍼 메모리에서 유지되므로, Hybrid TOE 전용의 최적화된 소켓 버퍼 관리 알고리즘이 필요하다. 따라서, ①과 ⑦을 담당하는 소켓 버퍼 관리 모듈을 하드웨어로 구현하면 프로토콜 처리 시간을 줄이는 데 중요한 역할을 담당할 것으로 기대된다. ②와 ⑥은 주메모리에 위치하는 사용자 영역과 TOE 어댑터 상에 존재하는 소켓 버퍼 사이의 데이타 복사에 관련된 기능으로, PCI 버스에서 DMA(Direct Memory Access)를 통해 효율적으로 처리할 수 있다. ④에 대해서는 순서 번호 검사와 TCP 체크섬 검사 등의 기능을 하드웨어로 구현하고, ⑤에 대해서는 ACK 패킷을 생성하는 기능 등을 하드웨어로 구현한다.

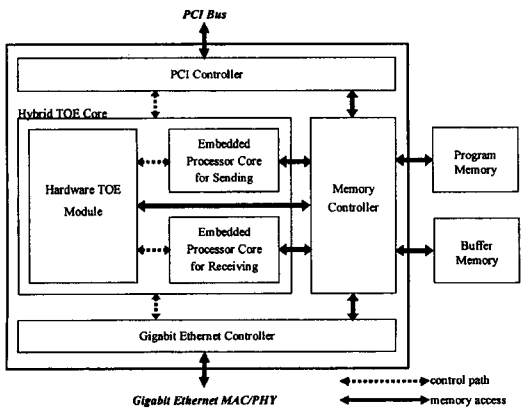

그림 4 Hybrid TOE 구현의 블록 다이어그램

③에 의해 발생하는 스케줄링의 부하는 다음과 같은 방식으로 해결할 수 있다. TCP/IP가 호스트 CPU에서 처리되는 상황을 살펴보면, 송신 프로세스와 수신 프로세스 이외에도 각종 사용자 프로세스들과 커널 프로세스들이 한정된 CPU 자원을 공유한다. 이들을 모두 지원하기 위해서 프로세스 스케줄러가 사용되고, 각 프로세스들 사이의 작업 전환 과정에서 오버헤드가 발생한다. 반면에 TCP/IP를 네트워크 어댑터에서 처리하는 경우에는 송신 프로세스와 수신 프로세스만 처리하면 된다. 따라서 Hybrid TOE 구조에서는 송신 프로세스와 수신 프로세스를 독립된 두 개의 내장형 프로세서에서 분리하여 처리하고, 이를 통해 스케줄링 오버헤드를 제거할 수 있다.

Hybrid TOE는 하드웨어 및 소프트웨어 구현을 함께 적용하였으므로, 대역폭 측면에서 소프트웨어 기반 TOE에 비해서는 우수한 성능을 나타낼 것이다. 또한, 성능에 큰 영향을 주는 기능들을 하드웨어로 구현하였으므로 하드웨어 기반 TOE에 근접하는 성능을 보일 것으로 예상된다. 또한, TCP/IP를 기반으로 하는 RDMA, iSCSI 등의 상위 프로토콜과 차세대 인터넷 프로토콜인 IPv6 등을 소프트웨어로 신속히 구현할 수 있을 것이다.

5. 결론 및 향후 과제

TOE는 네트워크 어댑터 상에서 TCP/IP를 처리함으로써 호스트 CPU에 가해지는 작업 부하를 줄이고 통신

의 성능을 높이는 기술이다. 본 논문에서는 효율적인 TOE 구조를 개발하기 위해 Linux 운영체제에 내장된 TCP/IP 프로토콜 스택을 분석하였고, 함수들의 수행시간을 측정한 실험을 통해 TCP/IP 내부에서 많은 부하를 발생시키는 주요한 요소들을 분석하였다. 실험 결과에 따르면 송신 과정에서는 소켓 버퍼를 확보하는 과정, 사용자의 데이타를 소켓 버퍼로 복사하는 과정, 그리고 프로세스 스케줄링 과정에서 호스트 CPU에 많은 부하가 발생하였다. 수신 과정에서는 TCP 헤더를 검사하고 처리하는 과정, ACK 처리 과정, 수신 데이타를 사용자 영역으로 복사하는 과정, 그리고 소켓 버퍼를 해제하는 과정에서 많은 부하가 발생하였다. 이러한 실험 결과를 바탕으로 본 논문에서는 하드웨어 기반 TOE 구현과 소프트웨어 기반 TOE 구현의 장점들을 취합한 Hybrid TOE 구조를 제안하였다. Hybrid TOE 구조에서는 통신의 성능에 큰 영향을 끼치는 기능들은 하드웨어로 구현하고 나머지 기능들은 소프트웨어로 구현함으로써 통신의 성능과 개발의 효율성을 동시에 제공할 수 있다.

향후 과제로는 본 논문에서 제안한 Hybrid TOE 구조를 구현하고, 이를 탑재한 네트워크 어댑터를 개발하여 Hybrid TOE의 동작과 성능을 검증할 계획이다.

참 고 문 헌

[1] Bierbaum, N., "MPI and Embedded TCP/IP Gigabit Ethernet Cluster Computing," Proceedings of 27th Annual IEEE Conference on Local Computer Networks 2002 (LCN 2002), pp. 733-734, Nov. 2002.

[2] Kay, J. and Pasquale, J., "Profiling and reducing processing overheads in TCP/IP," IEEE/ACM Transactions on Networking, Vol. 4, No. 6, pp. 817-828, Dec. 1996.

[3] Camarda, P., Pipio, F. and Piscitelli, G., "Performance evaluation of TCP/IP protocol implementations in end systems," IEE Proceedings of Computers and Digital Techniques, Vol. 146, No. 1, pp. 32-40, Jan. 1999.

[4] Zec, M., Mikuc, M. and agar, M., "Estimating the Impact of Interrupt Coalescing Delays on Stedy State TCP Throughput, Interrupt Coalescing," Proceedings of International Conference on Software, Telecommunications and Computer Networks, Oct. 2002.

[5] "Intel PRO/1000T IP Storage Adapter," Data Sheet, http://www.intel.com/network/connectivity/resources/doc_library/data_sheets/pro1000_T_IP_SA.pdf, Intel, 2003.

[6] Satran, J., Meth, K., Sapuntzakis, C., Chadalapaka, M. and Zeidner, E., "Internet Small Computer Systems Interface (iSCSI)," IETF RFC 3720, http://www.ietf.org/rfc/rfc3720.txt, April 2004.

[7] Aiken, S., Grunwald, D., Pleszkun, A. R. and Willeke, J., "A Performance Analysis of the iSCSI Protocol," Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS '03), 2003.

[8] "SLIC Technology Overview," http://www.alacritech.com/html/tech_review.html, Alacritech, 2002.

[9] "iSCSI Controller," Data Sheet, http://download.qlogic.com/datasheet/16291/isp4010.pdf, qLogic, 2003.

[10] "Adaptec TOE NAC 7711," Data Sheet, http://graphics.adaptec.com/pdfs/ana_7711_datasheet.pdf, Adaptec, 2003.

[11] "Alacritech SES1001T: iSCSI HBA Competitive Analysis," Benchmark Report, http:// www.veritest.com/clients/reports/alacritech/alac_ses1001t.pdf, VeriTest, March 2004.

[12] "Unleashing File Server Potential with Adaptec GigE NAC 7711," Benchmark Report, http://graphics.adaptec.com/pdfs/NAC_appbrief.pdf, Adaptec, 2003.

[13] Deering, S. and Hinden, R., "Internet Protocol, Version 6 (IPv6) Specification," IETF RFC 2460, http://www.ietf.org/rfc/rfc2460.txt, Dec. 1998.

[14] Recio, R., Culley, P., Garcia, D. and Hilland, J., "An RDMA Protocol Specification (Version 1.0)," http://www.rdmaconsortium.org/home/draft-recio-iwarp-rdmap-v1.0.pdf, RDMA Consortium, Oct. 2002.

[15] Cox, A., "Network Buffers and Memory Management," Linux Journal, Issue 29, Sep. 1996.

장 한 국

1999년 부산대학교 컴퓨터공학과 학사 2001년 부산대학교 컴퓨터공학과 석사 수료. 2001년~현재 부산대학교 컴퓨터공학과 석박사 통합과정. 관심분야는 컴퓨터구조, 클러스터시스템, TOE, SAN

오 수 철

1995년 부산대학교 컴퓨터공학과 학사 1997년 부산대학교 컴퓨터공학과 석사 1997년~1998 LG전자 멀티미디어 연구소 연구원. 1998년~2003 부산대학교 컴퓨터공학과 박사. 2003년~2004년 (주)아이온테크 연구원. 2004년~현재 부산대학교 BK21 사업단 기금교수. 관심분야는 클러스터 시스템, 병렬처리, CC-NUMA, VOD

정 상 화

1985년 서울대학교 전기공학과 학사 1988년 Iowa State University 전기 및 컴퓨터공학과 석사. 1993년 University of Southern California 전기및컴퓨터공학과 박사. 1993년~1994년 University of Central Florida 전기및컴퓨터공학과 조교수. 1994년~2000년 부산대학교 컴퓨터공학과 조교수 2000년~현재 부산대학교 컴퓨터공학과 부교수, 컴퓨터및정보통신연구소 연구원. 관심분야는 클러스터시스템, 병렬처리, VIA, VOD, TOE, RDMA

김 동 규

1992년 서울대학교 컴퓨터공학과 학사 1994년 서울대학교 컴퓨터공학과 석사 1999년 서울대학교 컴퓨터공학과 박사 1999년~2001년 부산대학교 기금조교수 대우. 2001년~현재 부산대학교 컴퓨터공학과 조교수. 관심분야는 보안, 정보보호, TOE, IPSec