멀티 코어 시스템을 위한 고속 노드내 통신 지원 모듈

(A Kernel Module to Support High-Performance Intra-Node Communication for Multi-Core Systems)

진 현 욱 † 강 현 구 †† 김 종 순 ††
(Hyun-Wook Jin) (Hyun-Goo Kang) (Jong-Soon Kim)

요 약 병렬 클러스터 컴퓨팅 시스템에서는 노드간의 효율적인 통신이 시스템의 전체 성능을 좌우하는 중요한 요소로 인식되어 왔다. 따라서 지금까지의 많은 연구들은 노드간 통신(inter-node communication)의 성능 향상에 초점을 맞췄다. 하지만 최근 등장한 멀티 코어 프로세서(multi-core processor)는 노드간 통신 외에도 노드내 통신(intra-node communication)의 중요성을 크게 부각시키고 있다. 이와 같이 그 중요성이 점점 더 증가하고 있는 노드내 통신의 성능을 향상시키기 위해서 여러 가지 노드내 통신 향상 기법들이 제안되어 왔다. 본 논문에서는 운영체제 커널의 도움으로 노드내 통신 시 발생하는 데이타 복사를 최소화하는 기법을 제안한다. 제안된 기법은 프로세스의 통신 버퍼를 상대 프로세스의 메모리 영역에 매핑하여 데이타 복사가 한번만 발생하도록 한다. 특히 제안된 기법은 리눅스 커널 버전 2.6을 위해서 설계된다. 성능 측정은 멀티 코어 프로세서를 장착한 시스템에서 이루어 졌으며, 기존 구현과 비교하여 본 논문에서 구현된 커널 모듈이 중간 및 작은 데이타 크기에 대해서 지연시간과 처리율을 각각 최대 62%와 144% 향상시킴을 보인다. 또한 프로세스가 수행되는 코어의 위치에 따라서 다른 성능을 보일 수 있음을 보인다.

키워드 : 클러스터, 노드내 통신, 멀티 코어, 메모리 매핑, 리눅스

Abstract In parallel cluster computing systems, the efficiency of communication between computing nodes is one of important factors that decide overall system performance. Accordingly, many researchers have studied on high-performance inter-node communication. The recently launched multi-core processor, however, increases the importance of intra-node communication as well because the more the number of cores in a node, the more the number of parallel processes running in the same node. Though there have been studies on intra-node communications, these have limited considerations on the state-of-the-art systems. In this paper, we propose a Linux kernel module that minimizes the number of data copy by exploiting the memory mapping mechanism for high-performance intra-node communication. The proposed kernel module supports the Linux kernel version 2.6. The performance measurements over a multi-core system present that the proposed kernel module can achieve lower latency up to 62% and higher throughput up to 144% than an existing kernel module approach. In addition, the measurements reveal that the performance of intra-node communication can vary significantly based on whether the cores that run the communication processes are belong to the same processor package (i.e., sharing the L2 cache).

Key words : Cluster, Intra-Node Communication, Multi-Core, Memory Mapping, Linux

· 본 논문은 2006년도 건국대학교 신임교원연구비와 한국전자통신연구원 위탁과제(#6010-2007-0022) 지원에 의한 것임
† 종신회원 : 건국대학교 컴퓨터공학부 교수
jinh@konkuk.ac.kr
†† 비 회 원 : 건국대학교 컴퓨터공학부
dosung51@naver.com
rebistence@naver.com
논문접수 : 2007년 6월 13일
심사완료 : 2007년 7월 22일

1. 서 론

클러스터 구조는 이미 고성능 컴퓨팅을 위한 서버 구조로 널리 사용되고 있다. 클러스터는 여러 개의 컴퓨팅 노드들이 고속의 네트워크로 연결되어 있는 구조이기 때문에 노드간의 효율적인 통신이 시스템의 전체 성능을 좌우하는 중요한 요소로 인식되어 왔다[1-6]. 하지만

최근의 주목할 만한 클러스터 구조의 진화는 노드간 통신(inter-node communication) 외에도 노드내 통신(intra-node communication)의 중요성을 크게 부각시켰다. 다중 코어 프로세서(multi-core processor)[7,8]의 등장과 개인용 슈퍼컴퓨터[9,10]의 등장이 그 진화의 대표적인 예라고 할 수 있다. 다중 코어 프로세서는 서로 독립적인 여러 개의 프로세서 코어가 하나의 집적 회로에 통합되어 있는 새로운 프로세서 구조다. 이러한 다중 코어 프로세서는 하나의 컴퓨팅 노드에 장착되는 컴퓨팅 코어의 개수를 크게 증가시키는 결과를 가져왔다. 또한 그로 인해서 개인용 슈퍼컴퓨터의 등장이 가능하게 되었다.

병렬 응용프로그램은 일반적으로 병렬성의 극대화를 위해서 시스템에 장착되어 있는 프로세서의 개수만큼 병렬 프로세스를 생성하여 실행한다. 이들 병렬 프로세스들은 서로 데이타 교환 및 동기화 등을 위해서 통신을 수행한다. 이 때 통신에 참여하는 두 프로세스가 서로 다른 컴퓨팅 노드에서 실행중이면 노드간 통신을 수행하며, 동일한 노드에서 실행중이면 노드내 통신을 수행하게 된다. 따라서 하나의 컴퓨팅 노드에 장착된 코어의 개수가 증가하면 노드내 통신의 비중이 증가한다. 병렬 프로세스간의 통신 지역성(locality)을 고려한다면 노드내 통신의 비중이 더 커진다고 할 수 있다.

이와 같이 그 중요성이 점점 더 증가하고 있는 노드내 통신의 성능을 향상시키기 위해서 여러 가지 노드내 통신 향상 기법들이 제안되어 왔다[11-14]. 노드내 통신을 기존의 IPC(Inter-Process Communication)를 사용하여 수행할 수도 있으나, 병렬 프로세스들은 메시지 전달을 위한 미들웨어 또는 라이브러리를 사용하기 때문에 IPC를 사용해도 여러 번의 데이타 복사가 발생한다. 고속 병렬 응용프로그램이 처리해야 하는 데이타의 크기가 크게 증가하고 있는 추세를 고려해 볼 때 노드내 통신 시 발생하는 데이타 복사를 최소화하는 것이 특히 중요하다. 이러한 데이타 복사 최소화를 위해서 메모리 매핑을 사용하는 기법들이 제시되었으며 이들은 대부분 클러스터에서 가장 많이 사용되고 있는 운영체제인 리눅스[15]를 대상으로 하고 있다. 하지만 이들 기법들은 리눅스 커널 버전 2.4 또는 그 이하 버전에서만 동작된다. 현재 클러스터를 구성하는 리눅스 운영체제가 커널 버전 2.6 기반으로 이동하고 있는 것을 고려해 볼 때, 커널 기반의 노드내 통신 기법들도 그에 따른 설계 및 구현의 변화가 필수적이다. 또한 기존의 기법들은 멀티 코어 환경에서 성능 분석이 이루어지지 않아서 멀티 코어 시스템에서 발생할 수 있는 특성에 대한 고찰이 부족하다.

본 논문에서는 이러한 기존 고속 노드내 통신 기법들의 한계를 극복하기 위해서 리눅스 커널 버전 2.6을 위한 노드내 통신 지원 커널 모듈을 제안한다. 제안된 기법은 프로세스의 통신 버퍼를 상대 프로세스의 메모리 영역에 매핑하여 데이타 복사가 한번만 발생하도록 한다. 이를 위해서 리눅스 커널 버전 2.6의 특성을 고려하여 메모리 매핑, 자료구조의 접근 동기화 등을 설계한다. 성능 분석을 위해서 멀티 코어 프로세서를 장착한 시스템에서 통신 지연시간과 처리율을 측정한다. 측정 결과는 기존 구현과 비교하여 본 논문에서 구현된 커널 모듈이 중간 및 작은 데이타 크기에 대해서 성능을 향상시킴을 보인다. 또한 송수신 프로세스가 같은 프로세서에 속해있는 코어에서 수행될 때와 다른 프로세서에 속해있는 코어에서 수행될 때 보이는 성능 차이를 제시한다.

본 논문은 서론에 이어 2장에서 연구의 배경이 되는 기존 연구들을 기술한다. 3장에서는 리눅스 커널 버전 2.6을 위한 고속 노드내 통신 지원 커널 모듈의 설계 및 구현을 제안한다. 4장에서는 구현된 커널 모듈의 성능을 측정한다. 마지막으로 5장에서 본 논문의 결론을 맺는다.

2. 연구 배경

본 장에서는 우선 노드내 통신을 위해서 기존 IPC를 사용했을 때의 한계에 대해서 논의한다. 그리고 노드내 통신과 관련된 기법 및 연구들에 대해서 기술한다. 마지막으로 본 연구에서 제안하는 커널 모듈의 기본 설계로 사용되는 LiMIC에 대해서 설명한다.

2.1 IPC(Inter-Process Communication)의 한계

노드내 통신을 수행하기 위해서는 기존의 여러 가지 IPC 기법들을 사용할 수 있다. 예를 들어 널리 사용되는 System V IPC의 일종인 공유 메모리(shared memory)는 여러 프로세스 간에 특정 메모리 영역을 공유함으로써 데이타 이동시 성능을 크게 향상 시킬 수 있다. 하지만 병렬 프로세스들은 메시지 전달을 위해서 노드간 통신과 노드내 통신을 구분하지 않고 미들웨어 또는 라이브러리가 제공하는 인터페이스를 사용한다. 예를 들어서, MPI(Message Passing Interface) [16] 병렬 응용프로그램은 MPI_Send( ), MPI_Isend( ), MPI_Recv( ), MPI_Irecv( )와 같은 MPI가 제공하는 API를 사용하여 점대점(Point-to-Point) 통신을 수행한다. 따라서 노드내 통신을 위해서 공유 메모리를 사용하려면 MPI API 함수 내부에서 공유 메모리를 관리해야 한다. 그 결과 송신측의 사용자 버퍼에서 공유 메모리로, 그리고 공유 메모리에서 수신측 사용자 버퍼로 두 번의 데이타 복사가 발생한다. 이외에도 공유 메모리는 공유 버퍼의 크기가 제한적이라는 단점 또한 존재한다. 이와 같이 IPC는 멀티 프로세스 프로그래밍에서는 효율

적인 프로세스 간 통신을 지원하지만 병렬 프로세스들을 위해서는 한계를 갖고 있다.

2.2 기존 노드내 통신 기법

MPI 노드내 통신을 위해서 제안된 기법들은 크게 세 가지로 구분할 수 있다: i) 네트워크 프로토콜 수준의 결정, ii) 파일 기반의 공유 메모리, iii) 커널 수준의 메모리 매핑. 이들 기법들은 그림 1에 비교되어 있다.

네트워크 프로토콜 수준 결정(그림 1(a))의 경우, 통신 미들웨어 또는 라이브러리는 노드간 통신과 노드내 통신을 구분하지 않고 통신 프로토콜에게 전송 요청을 한다. 그 후 네트워크 프로토콜이 비로소 수신자가 동일 노드에 있는지를 판단하고 그렇다면 노드내 통신을 수행한다. 이 때 네트워크 프로토콜이 호스트 측에 구현되어 있다면 (예, 커널에 구현되어 있는 IP) 두 번의 복사가 발생하며, NIC(Network Interface Card)에 구현되어 있다면 (예, 사용자 수준 프로토콜, 프로토콜 오프로드 엔진) 두 번의 DMA(Direct Memory Access)가 발생한다.

파일 기반의 공유 메모리 기법(그림 1(b))은 System V IPC의 일종인 공유 메모리 기법과 비슷한 방식이다 [11]. 동일한 노드에 속하는 프로세스들은 공유 메모리를 할당하고 이를 통해서 통신한다. 일반적으로 공유 메모리는 파일을 메모리 매핑하여 생성한다. 이러한 공유 메모리의 생성 기법은 System V IPC의 공유 메모리에 비해서 큰 버퍼를 생성할 수 있는 장점이 있다. 하지만 노드내 통신을 위해서는 여전히 두 번의 데이타 복사가 발생한다. 파일 기반의 공유 메모리 기법은 위에서 언급한 네트워크 프로토콜 수준의 결정 보다 일반적으로 좋은 성능을 보인다.

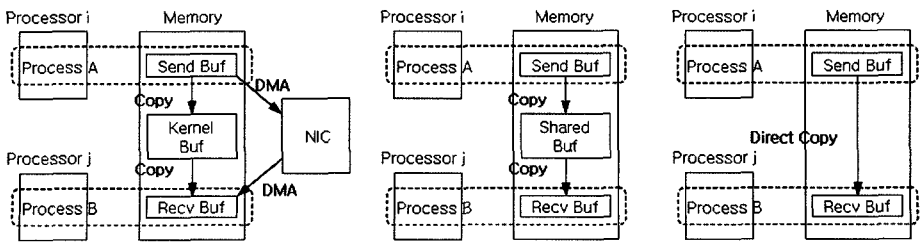
커널 수준의 메모리 매핑 기법(그림 1(c))은 노드내 통신 시 발생하는 데이타 복사를 한 회로 줄인다 [12-14]. 이 기법은 커널의 도움으로 상대 프로세스의 버퍼를 다른 프로세스의 메모리로 매핑시킨 후 복사를 수행함으로써 노드내 통신을 완료한다. 따라서 한 회의 데이타 복사만 발생한다. 그 결과 커널 수준의 메모리 매핑 기법은 대부분의 데이타 크기에 대해서 앞에서 언급된 두 기법에 비해서 좋은 성능을 보인다. 하지만 작은 크기의 데이타에 대해서는 파일 기반의 공유 메모리 기법이 더 낮은 지연시간을 보인다. 이것은 작은 크기의 데이타에 대해서는 커널 수준의 메모리 매핑 오버헤드가 제거된 데이타 복사 오버헤드에 비해서 상대적으로 크기 때문이다 [17].

2.3 LiMIC 개요

고속 병렬 응용프로그램이 처리해야 하는 데이타의 크기가 크게 증가하고 있는 추세를 고려해 볼 때, 큰 데이타에 대해서 월등히 좋은 성능을 보여주는 커널 수준의 메모리 매핑 기법들은 향후 더욱 주목 받을 것으로 기대된다. 본 장에서는 특히 최근에 제안된 LiMIC[14]에 대해서 설명한다. 3장에서 제안되는 노드내 통신 지원 커널 모듈은 LiMIC을 그 기본 설계로 한다.

LiMIC은 기존의 커널 수준 메모리 매핑 기법들과는 다르게 특정 네트워크 연결(network interconnection)에 의존적이지 않도록 설계되었다. 따라서 모든 종류의 네트워크 연결에 대해서 적용될 수 있다. 또한 MPI에 친숙한 인터페이스를 제공함으로써 여러 종류의 MPI 구현에 적용되어 노드내 통신의 성능을 향상시킬 수 있다. 또한 기존의 제안들과는 다르게 데이타 복사를 수신과 송신 프로세스가 모두 수행 가능하다. 그리고 LiMIC은 커널 모듈로서 필요한 모든 기능을 구현하도록 설계되어 커널의 수정 없이 필요한 메모리 매핑 및 데이타 이동을 수행할 수 있다. 이러한 장점들은 LiMIC이 리눅스 클러스터 시스템들에 널리 사용될 수 있게 할 것으로 기대된다. 그 예로서 LiMIC은 InfiniBand[18]를 위한 MPI 구현인 MVAPICH[19]에 적용되어 병렬 응용프로그램의 성능을 향상 시킬 수 있음을 보였다.

LiMIC은 리눅스 커널 버전 2.4 환경을 대상으로 설계되었다. LiMIC은 메모리 매핑을 위해서 kiobuf를 사용한다. kiobuf는 가상 메모리 시스템의 복잡함을 추상화하여 메모리 매핑 등의 복잡한 메모리 조작 작업들을 쉽게 처리할 수 있도록 한다. 하지만 kiobuf는 자료 구



(a) 네트워크 프로토콜 수준의 결정 (b) 파일 기반의 공유 메모리 (c) 커널 수준의 메모리 매핑

그림 1 노드내 통신 기법 분류

조를 할당할 때 큰 오버헤드가 발생하여 이를 해결하기 위해서 LiMIC은 커널 모듈 초기화시에 kiobuf 자료 구조를 미리 다수 할당하여 pool을 생성한다. kiobuf는 원래 리눅스 커널 버전 2.4에서 입출력 장치의 직접 입출력(direct I/O)을 위해서 제안되었다. 이러한 kiobuf는 커널 버전 2.6에서는 더 이상 지원되지 않는다. 따라서 커널 버전 2.6 환경을 위해서는 LiMIC의 메모리 매핑 구현이 수정되어야 한다.

동일 노드에서 수행중인 프로세스들은 통신을 위해서 LiMIC에게 동시에 통신 요청할 수 있다. 이러한 경우를 위해서 LiMIC은 커널 모듈 내부의 공유 자료 구조들에 대해서 spin_lock( ) 기반의 접근 동기화를 수행한다. 주의할 중요한 사항은 spin_lock( )에 의해서 보호되는 임계 영역 내부에서는 프로세스 스케줄러가 호출되어서는 안 된다는 것이다. 하지만 커널 버전 2.6에서 구현이 변경된 커널 함수들은 LiMIC이 이러한 조건을 더 이상 만족시키지 못하게 한다. 따라서 임계 영역에 대한 수정이 역시 요구된다.

그리고 아직까지 LiMIC을 포함한 기존의 커널 수준 메모리 매핑 기법들은 멀티 코어 시스템에서 그 성능 측정 및 분석이 아직 이루어지지 않았다.

3. 설계 및 구현

본 장에서는 리눅스 커널 버전 2.6 환경을 위한 커널 수준 메모리 매핑 기법을 설계한다. 제시된 설계는 2.3장에서 언급한 LiMIC의 장점들을 갖는다.

3.1 메모리 매핑 기반의 데이타 복사 제거

기존의 LiMIC에서 사용되었던 kiobuf는 더 이상 리눅스 커널 버전 2.6에서는 지원되지 않는다. 따라서 메모리 매핑을 위해서는 여러 가지 커널 함수의 호출 및 자료 구조 조작을 직접 해야 한다. 대표적인 예로서 메모리 매핑 하려는 사용자 메모리 영역의 페이지 설명자(page descriptor)를 얻기 위해서 get_user_pages( ) 함수를 호출해야 한다.

그림 2는 리눅스 커널 버전 2.6이 제공하는 함수들을 사용하여 메모리 매핑과 데이타 이동을 수행하는 과정을 보여준다. 우선 사용자 프로세스는 통신 라이브러리의 API를 호출하고, 통신 라이브러리는 이 통신이 노드내 통신인지를 판단한다(그림 2의 1). 노드내 통신일 경우에는 LiMIC이 제공하는 인터페이스를 통해서 (그림 2의 2) ioctl( ) 시스템 호출(system call)을 사용하여 커널 모듈에게 사용자 버퍼 정보를 전달한다(그림 2의 3). 커널 모듈은 통신 요청을 큐(queue)에 넣고 ioctl( )을 종료한다(그림 2의 4).

이후에 상대방 프로세스의 경우는 통신 라이브러리의 API를 호출하면(그림 2의 5), LiMIC의 인터페이스를 통하고(그림 2의 6과 7), 커널 모듈에 의해서 해당 통신 요청을 큐에서 검색한다(그림 2의 8). 검색된 통신 요청에 저장되어 있는 사용자 버퍼의 페이지 설명자를 얻기 위해서 get_user_pages( ) 함수를 호출하고, 획득된 페이지 설명자를 기반으로 kmap( ) 커널 함수를 사용하여 현재 프로세스의 가상 메모리에 매핑을 수행한다(그림 2의 9). 마지막으로 현재 프로세스가 수신측이라면 copy_to_user( ) 커널 함수를 사용하여 매핑된 메모리 영역으로부터 현재 프로세스의 사용자 영역으로 복사를 수행하여 수신을 완료한다. 반대로 현재 프로세스가 송신측이라면 copy_from_user( )를 사용하여 현재 프로세스의 사용자 영역으로부터 매핑된 메모리 영역으로 복사를 수행하여 송신을 완료한다(그림 2의 10).

이와 같이 수정된 LiMIC의 설계는 기존 LiMIC의 장점을 그대로 유지시킬 수 있다. 우선 메모리 복사를 한 회를 줄임으로써 대용량 데이타 통신의 성능을 향상시킨다. 또한 커널의 수정을 요구하지 않으므로 커널 모듈만으로도 필요한 메모리 매핑 기능을 구현할 수 있다.

3.2 커널 자료 구조의 접근 동기화

노드내 통신을 위한 커널 모듈은 3.1장에서 언급되었

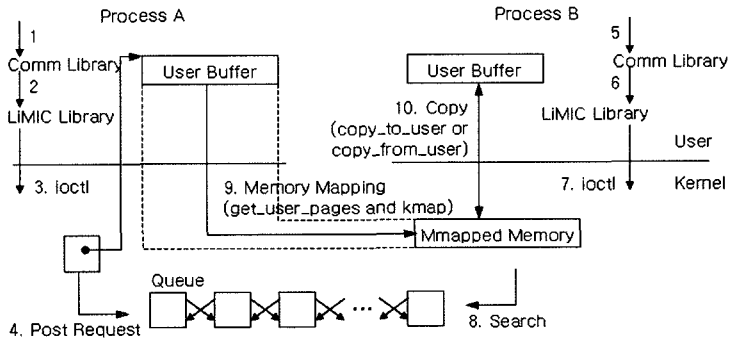


그림 2 메모리 매핑 및 데이타 이동

듯이 여러 프로세스들로부터 ioctl( ) 시스템 호출을 통해서 요청을 받는다. 이러한 요청들은 서로 병렬적으로 발생할 수 있으며, 이들을 순차 처리한다면 성능의 병목 지점이 될 수 있다. 이를 해결하고 노드내 여러 프로세스들의 요청들을 동시에 처리하기 위해서는 i) 커널 동시 진입의 허가와 ii) 공유 자료 구조의 접근 동기화가 해결되어야 한다.

3.1장에서 언급된 ioctl( ) 시스템 호출은 내부적으로 커널 모듈의 file_operations 구조체에 있는 해당 함수 포인터에 의해서 구현된다. 리눅스 커널 버전 2.6에서는 이를 위한 함수 포인터가 ioctl과 unblocked_ioctl이라는 이름으로 두 개가 존재한다. 기존 커널 버전 2.4 이하에서는 ioctl만 존재했는데, 이는 현재 BKL(Big Kernel Lock) 상태로 수행되는 ioctl을 위해서 사용된다. 즉, 이 경우는 여러 프로세스가 동시에 커널로 진입하는 것이 불가능하다. 반면에 unblocked_ioctl은 이를 허용한다. 따라서 프로세스들의 요청을 커널이 동시에 처리 가능하다. 하지만 내부 공유 자료 구조에 대한 접근 동기화를 자체적으로 해결해 줘야 한다.

노드내 통신을 위해서 설계된 커널 모듈 내부에서 공유되는 대표적인 자료 구조는 큐다. 3.1장에서 언급되었듯이 여러 프로세스들은 통신 요청 정보를 저장 또는 검색을 하기 위해서 큐를 접근한다. 따라서 큐를 접근하는 부분만을 임계 영역으로 설계하면 여러 프로세스들의 커널 동시 접근은 가능해진다. 하지만 이 경우 역시 커널 내부에서의 병렬성을 극대화하기에는 부족하다. 이러한 한계를 극복하기 위해서 제안된 커널 모듈은 하나의 큐만을 생성하지 않고 각 프로세스마다 세 개의 큐를 별도로 생성하도록 한다. 이들 세 큐는 각각 송신 요청, 수신 요청, 송수신 완료를 저장하기 위해서 사용한다. 그리고 동시 접근의 제어는 이들 큐 단위로 별도 수행함으로써 임계 영역을 작고 분산되도록 한다.

큐들의 접근 동기화를 위해서는 spin_lock( )을 사용한다. 세마포어(semaphore)를 사용하여 동기화를 구현할 수도 있으나 병렬 서버에서는 가용 프로세서의 개수만큼의 프로세스를 생성하기 때문에 spin_lock( )이 더 좋은 응답 시간을 줄 수 있다. 반면 세마포어는 임계 영역에 진입을 실패하게 되면 프로세스가 잠들 수 있기 때문에 응답 시간이 길어질 수 있다. spin_lock( )의 중요한 특징 중 하나는 spin_lock( )에 의한 임계 영역 내에서는 프로세스 스케줄링이 발생되면 안 된다는 것이다. 이것은 교착 상태(deadlock)가 발생하는 것을 방지하기 위해서다. 리눅스 커널 버전 2.6의 get_user_pages( ) 함수는 커널 버전 2.4에서와는 다르게 스케줄링 함수를 호출한다. 따라서 get_user_pages( ) 함수를 호출하는 부분은 임계 영역에서 제외를 시켜야 한다. 이러한 조건들을 모두 만족하도록 spin_lock의 구조를 재구성하였다. 이 과정에서 그림 2와 같이 get_user_pages( ) 함수는 메모리 매핑을 수행하는 프로세스가 호출하도록 되었다. 기존 LiMIC은 매핑될 사용자 버퍼의 페이지 설명자를 얻는 프로세스와 실제 메모리 매핑을 수행하는 프로세스가 달랐다.

4. 성능 측정 결과

본 장에서는 3장에서 제안된 커널 수준 메모리 매핑 기법의 성능을 측정하고 분석한다. 우선 4.1장에서는 성능 측정 방법에 대해서 설명한다. 이후에 기존의 커널 버전 2.4를 위해서 개발된 LiMIC과 3장에서 제안된 설계를 기반으로 구현된 모듈의 성능을 비교한다. 그리고 마지막으로 본 논문에서 개발된 모듈을 기반으로 멀티 코어 시스템에서 관찰할 수 있는 성능 특성을 분석한다.

4.1 성능 측정 방법

성능 측정은 단방향 통신 지연 시간(one-way latency)과 통신 처리율(throughput)에 대해서 수행된다. 단방향 통신 지연시간 측정을 위해서 전형적인 ping-pong 테스트 프로그램을 사용한다. ping-pong 테스트 프로그램의 두 프로세스는 서로 특정 크기의 메시지를 주고받으며 왕복 시간을 측정하고 이 값을 반으로 나누어 단방향 통신 지연시간을 계산한다. 본 논문에서는 4B부터 512KB까지 메시지 크기를 변화시켰으며 각 메시지 크기에 대해서 1000번의 측정 평균값을 사용하여 결과 값을 기록한다.

통신 처리율을 측정하기 위해서는 각각 송신과 수신 역할을 수행하는 프로세스를 생성한다. 송신 프로세스는 특정 크기의 메시지를 1000회 연속적으로 송신 요청하고 수신 프로세스는 이들을 모두 수신 완료하는데 걸리는 시간을 측정한다. 측정된 시간과 전송된 전체 메시지의 용량으로 통신 처리율을 측정한다. 통신 처리율 측정 역시 메시지 크기를 4B부터 512KB까지 변화시킨다.

4.2 리눅스 커널 버전에 따른 성능 비교

본 장에서는 리눅스 커널 버전 2.4를 위해서 구현된 LiMIC과 3장에서 제안된 커널 모듈의 성능을 측정하고 비교한다. 성능 측정의 목표는 리눅스 커널 버전 2.4를 위해서 구현된 모듈과의 커널 버전 2.6을 위해서 구현된 커널 모듈의 성능이 비슷한 것을 보이는 것이다. 2.3장에서 설명되었듯이 기존의 LiMIC은 kiobuf 자료 구조 할당에 필요한 오버헤드를 줄이기 위해서 자료 구조 pool을 생성한다. 실험 결과에서 기존 LiMIC의 성능은 이러한 pool을 사용할 때와 사용하지 않을 때로 구분해서 보인다. 성능 측정은 단방향 통신 지연 시간과 통신 처리율에 대해서 수행된다. 성능 측정을 위해서 사용된 서버는 표 1과 같이 구성되었다.

표 1 리눅스 커널 버전에 따른 성능 측정 시스템

프로세서	두 개의 Intel Xeon 2.8GHz 프로세서 (2-Way Paxville Dual-Core)	
메모리	2GB (DDR2-400)	
시스템 버스	800MHz	
칩셋	Intel E7520 Chipset	
운영체제	기존 LiMIC	RedHat Enterprise Linux 3 (커널 버전 2.4.21)
	제안된 커널 모듈	Fedora Core 5 (커널 버전 2.6.15)

4.2.1 통신 지연시간

4.1장에서 언급되었듯이 ping-pong 테스트 프로그램을 사용하여 통신 지연시간을 측정하였다. 그림 3과 4는 그 실험 결과를 보여준다. 그림 3에서 볼 수 있듯이 작은 메시지 크기에 대해서 본 논문에서 제안된 커널 모듈(그림 3의 범례 "2.6")이 더 낮은 지연시간을 보이는 것을 알 수 있다. 그 결과 kiobuf 자료 구조 pool을 사용하는 기존 LiMIC(그림 3의 범례 "2.4 with pool")에 비해서 38%까지 지연시간을 낮췄으며, pool을 사용하지 않는 기존 LiMIC(그림 3의 범례 "2.4 without pool")에 비해서는 62%까지 지연시간을 낮췄다. 이와 같이 작은 메시지에 대해서 지연시간을 줄일 수 있는 것은 kiobuf 라는 추가적인 계층을 사용하지 않고 직접 필요한 커널 함수만을 호출하여 불필요한 오버헤드가 제거되었기 때문이다. 하지만 이와 같이 제거된 오버헤드는 메시지 크기가 증가함에 따라서 그림 4에서와 같이 크게 관찰되지 않는다. 이것은 큰 메시지의 경우에는 데이타 복사 오버헤드가 대부분의 지연시간을 차지하기 때문이다.

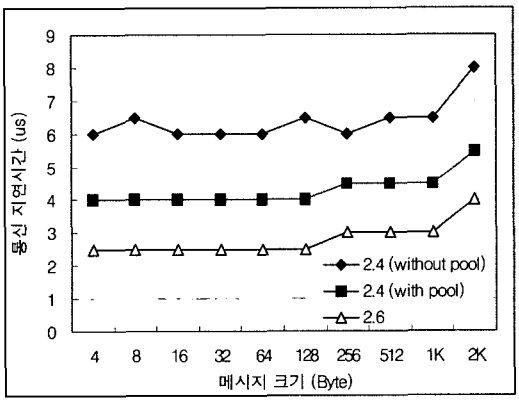


그림 3 리눅스 커널 버전에 따른 통신 지연시간(4B-2KB)

4.2.2 통신 처리율

그림 5와 6은 통신 처리율 측정 결과를 보여준다. 그림 5는 통신 지연시간과 같이 통신 처리율에서도 본 논문에서 제안된 커널 모듈이 작은 메시지 크기에 대해서 더 좋은 성능을 보여주고 있음을 알려준다. 그림에서 볼 수 있듯이 kiobuf 자료 구조 pool을 사용하는 기존 LiMIC에 비해서 89%까지 처리율을 높였으며, pool을 사용하지 않는 기존 LiMIC에 대해서는 144%까지 처리율을 높였다. 하지만 큰 메시지에 대해서는 그림 6과 같이 큰 성능 향상이 관찰되지 않는다. 오히려 일부 메시지 크기에 대해서는 pool을 사용하는 기존 LiMIC에 비해서 낮은 처리율을 보이고 있다. 이것은 3.2장에서 설명된 get_user_pages( ) 함수를 임계 영역 밖으로 옮긴 설계에 의해서 발생하는 것이다. 기존의 LiMIC은 페이지 설명자를 얻는 프로세스와 메모리 매핑을 수행하는 프로세스가 서로 달랐다. 하지만 페이지 설명자를 임계 영역 밖에서 구하도록 설계가 변형된 LiMIC에서는 동일 프로세스가 이 두 작업을 모두 수행하게 된다. 따라서 더 이상 이 두 작업은 서로 겹쳐서 수행되기 어렵다. 또한 이 두 작업은 메시지 크기(버퍼에 해당하는 페이지 개수)가 증가하면 그 오버헤드 또한 비례하여 증가

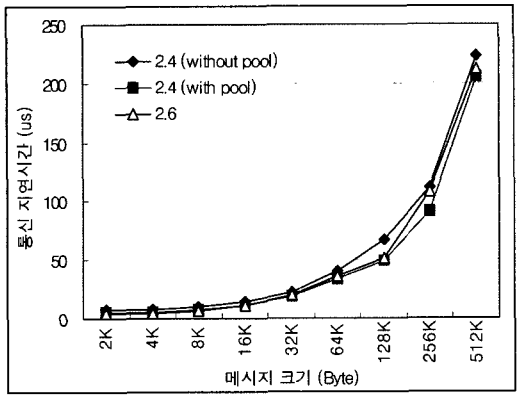


그림 4 리눅스 커널 버전에 따른 통신 지연시간 (2KB-512KB)

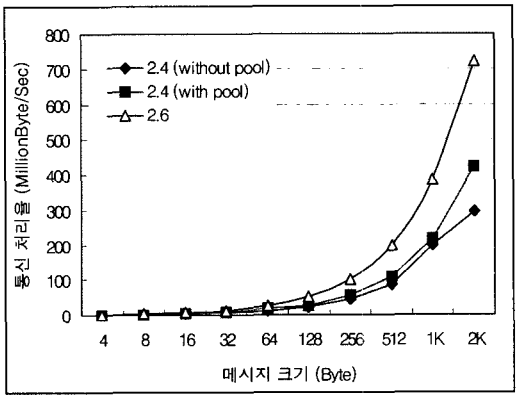


그림 5 리눅스 커널 버전에 따른 통신 처리율 (4B-2KB)

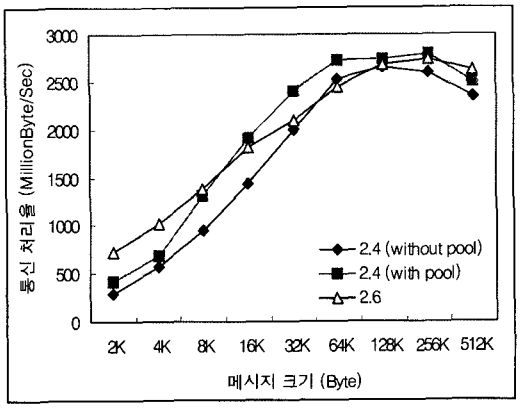


그림 6 리눅스 커널 버전에 따른 통신 처리율 (2KB-512KB)

한다. 따라서 큰 메시지에 대해서는 기존의 LiMIC보다 낮은 처리율을 보인다. 그림 6에서 256KB 이상의 메시지에 대해서 처리율이 모두 감소하는 것은 메시지의 크기가 프로세서 캐쉬의 용량을 벗어나서 데이타 복사 시에 캐쉬 효과를 얻지 못하기 때문이다.

4.3 멀티 코어 특성 분석

멀티 코어 시스템에서는 프로세스가 수행되는 코어에 따라서 성능이 달라질 수 있다. 예를 들어서, 송수신을 담당하는 프로세스가 모두 같은 프로세서에 속해 있는 코어에서 수행되는 경우와 서로 다른 프로세서의 코어에서 수행되는 경우 자원의 공유 여부에 따라서 다른 성능을 보일 수 있다. 이러한 성능의 차이를 관찰하기 위해서 본 논문에서 제안된 커널 모듈을 사용하는 송수신 프로세스를 다른 코어의 쌍에서 수행시킨 후 성능의 차이를 관찰한다.

이러한 관찰을 위해서 L2(Level 2) 캐쉬를 공유하는 멀티 코어 프로세서를 사용하여 측정 시스템을 표 2와 같이 구성한다. 4.2장에서 사용한 Paxville Dual-Core 프로세서는 하나의 프로세서 내에서도 서로 다른 코어는 2MB의 L2 캐쉬를 별도로 갖고 공유하지 않는다. 이와는 다르게 본 장에서 사용하는 Bensley(Woodcrest) Dual-Core 프로세서의 경우에는 하나의 프로세서에 포함되어 있는 코어들은 4MB의 L2 캐쉬를 공유한다.

실험 중에 통신에 참여하는 프로세스들은 sched_setaffinity( ) 시스템 호출을 사용하여 특정 코어에 프로세서 친화도를 정적으로 결정하도록 함으로써 인위적으로 프로세스가 수행되는 코어를 결정되도록 했다. 성능은 4.2장과 같이 단방향 통신 지연시간과 통신 처리율을 각각 측정한다.

표 2 멀티 코어 특성 측정 시스템

프로세서	두 개의 Intel Xeon 2.66GHz 프로세서 (2-Way Bensley (Woodcrest) Dual-Core)
메모리	2GB (DDR2-533)
시스템 버스	1333MHz
칩셋	Intel 5000P Chipset
운영체제	Fedora Core 5 (커널 버전 2.6.15)

4.3.1 통신 지연시간

그림 7과 8은 같은 프로세서에 포함되어 있는 코어에서 프로세스들이 수행되어 있는 경우(그림의 범례 "Cores in the same package")와 서로 다른 프로세서에 포함되어 있는 코어에서 프로세스들이 수행되었을 때(그림의 범례 "Cores in the different packages")의 통신 지연시간을 비교하고 있다. 그림 7에서 통신 프로세스들이 같은 프로세서에 포함되어 있는 코어에서 수행될 때 작은 크기의 메시지에 대해서 지연시간이 57%까지 감소하는 것을 관찰 할 수 있다. 또한 중간 크기인 8KB 메시지의 지연시간을 보면 36% 이상 감소한 것을 알 수 있다. 이러한 성능의 차이는, 노드내 통신을 위해

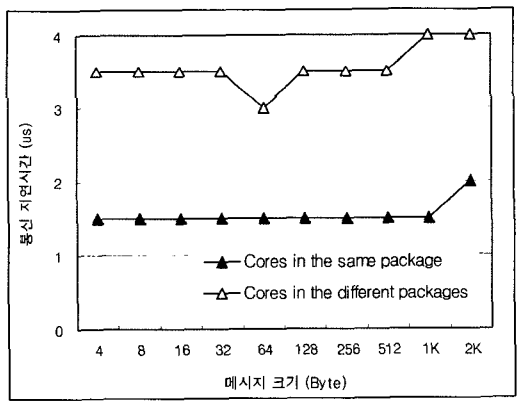


그림 7 프로세스 실행 코어에 따른 통신 지연시간 (4B-2KB)

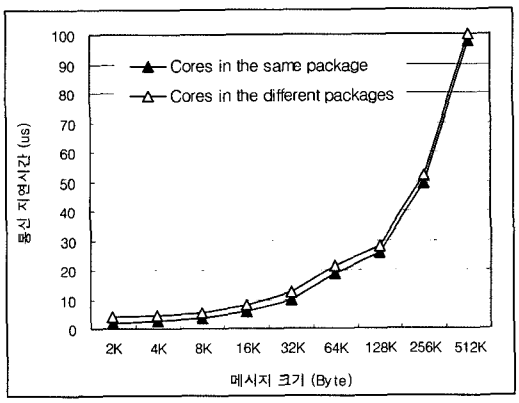


그림 8 프로세스 실행 코어에 따른 통신 지연시간 (2KB-512KB)

서 제안된 커널 모듈 내부에서 데이타 복사가 발생 할 때, 송수신 프로세스가 수행되는 코어가 L2 캐쉬를 공유할 경우 데이타 복사 오버헤드가 감소하기 때문이다. 하지만 큰 메시지의 경우에는 성능 차이의 폭이 크게 관찰되지 않음을 알 수 있다. 이것은 통신 지연시간 측정을 위해서 사용되는 ping-pong 테스트의 프로세스들이 모두 성능 측정 메시지 크기의 송신과 수신 버퍼를 갖고 있기 때문에 메시지 크기가 커지면서 캐쉬 효과를 크게 얻지 못함에 따라 발생하는 것으로 분석된다.

4.3.2 통신 처리율

그림 9와 10은 서로 다른 코어의 쌍에서 통신 프로세스들이 수행되었을 때의 통신 처리율 측정 결과를 비교하고 있다. 그림 9에서 관찰할 수 있듯이 통신 프로세스들이 같은 프로세서에 포함되어 있는 코어에서 수행될 때 2KB 이하의 메시지에 대해서 통신 처리율이 68%에서 100%까지 향상될 수 있음을 알 수 있다. 그리고 큰 메시지에 대해서도 통신 처리율이 눈에 띄게 차이가 나는 것을 그림 10에서 알 수 있다. 이러한 성능의 차이는 4.3.1장에서 언급되었듯이 프로세스를 실행시키는 코어 간 L2 캐쉬의 공유 여부에 따른 데이타 복사 오버헤드 감소에 의한 것이다. 통신 처리율도 4.3.1장에서 관찰한 통신 지연시간과 마찬가지로 메시지 크기가 커짐에 따라 L2 캐쉬의 공유 여부에 큰 영향을 받지 않는다. 하지만 통신 처리율 측정 프로세스의 경우, 송신 프로세스는 메시지 크기만큼의 송신 버퍼만을 갖고 있으며, 수신 프로세스는 메시지 크기만큼의 수신 버퍼만을 갖고 있기 때문에 지연시간 측정 프로세스들 보다는 적은 통신 버퍼를 관리하게 된다. 따라서 캐쉬의 장점을 더 큰 메시지에 대해서도 얻을 수 있다. 그 결과 지연시간 측정 결과 보다는 통신 처리율 측정 결과에서 큰 메시지에 대한 성능 차이를 쉽게 관찰할 수 있다.

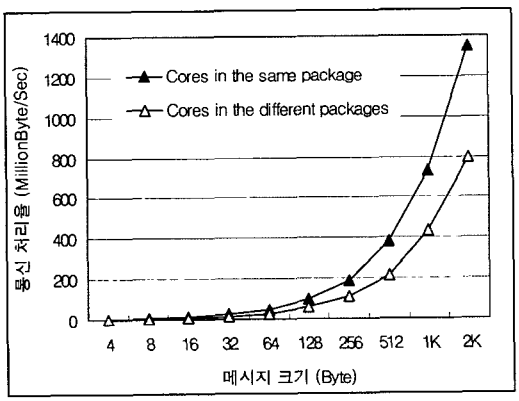


그림 9 프로세스 실행 코어에 따른 통신 처리율 (4B-2KB)

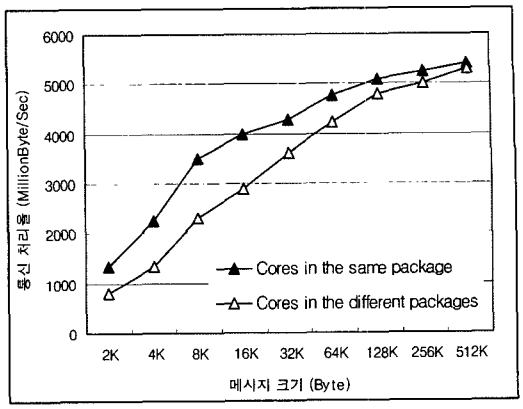


그림 10 프로세스 실행 코어에 따른 통신 처리율 (2KB-512KB)

5. 결론 및 향후 계획

본 논문은 멀티 코어 시스템에서 고성능 노드내 통신 지원을 위한 커널 모듈을 제안했다. 특히 기존의 연구들이 리눅스 커널 버전 2.4 또는 그 이하 버전을 대상으로 했던 것과는 다르게 본 연구에서 제안된 커널 모듈은 커널 버전 2.6을 대상 시스템으로 한다. 이를 위해서 리눅스 커널 버전 2.6의 특성을 반영하고 메모리 매핑과 자료구조의 접근 동기화 등의 설계를 새롭게 제안했다. 성능 측정 결과, 커널 버전 2.4에서 보인 기존 LiMIC의 성능과 비교했을 때, 제안된 커널 모듈이 중간 및 작은 데이타 크기에 대해서 지연시간과 처리율을 각각 최대 62%와 144% 향상시킴을 보였다. 현재 클러스터를 구성하는 리눅스 운영체제가 커널 버전 2.6 기반으로 이동하고 있는 것을 고려해 볼 때, 본 논문에서 제안된 커널 버전 2.6을 위한 커널 기반의 노드내 통신 지원 모듈은 큰 의미가 있다고 할 수 있다.

본 논문은 개발된 커널 모듈을 사용하여 멀티 코어 시스템에서 프로세스가 수행되는 코어에 따라서 발생할 수 있는 성능의 차이를 또한 보였다. 이러한 성능의 차이는, 제안된 커널 모듈 내부에서 데이타 복사가 발생할 때, 송수신 프로세스가 수행되는 코어가 L2 캐쉬를 공유할 경우 데이타 복사 오버헤드가 감소하기 때문이다. 그 결과 프로세스들이 같은 프로세서에 포함되어 있는 코어에서 수행될 때 통신 지연시간이 작은 크기의 메시지에 대해서는 57%까지 중간 크기 메시지에 대해서는 36% 이상 감소한 것을 알 수 있었다. 또한 통신 처리율의 경우는 2KB 이하의 메시지에 대해서 68%에서 100%까지 향상될 수 있음을 보였다. 이러한 측정 결과는 효율적인 노드내 통신을 위해서는 멀티 코어 시스템의 특성을 이해하고 그에 대한 연구가 필요함을 시사한다.

향후 제안된 커널 모듈을 사용하여 MPI 응용 프로그램 수준의 성능을 커널 버전 2.6 환경에서 측정할 예정이다. 이 외에도 MPI-2에서 정의되어 있는 One-Sided Communication의 노드내 수행을 효과적으로 지원하기 위한 방안 등을 연구할 계획이다. 이 외에도 본 논문에서 설계된 커널 모듈보다 경량의 노드내 통신 지원 모듈을 연구할 계획이다. 현재의 커널 모듈은 병렬 프로세스를 위한 미들웨어 또는 통신 라이브러리가 관리하는 메시지 큐 외에 별도의 메시지 큐를 내부적으로 관리하고 있다. 향후에는 이러한 추가적은 메시지 큐를 제거하고 노드내 통신을 위한 프리미티브만을 제공하여 효율성을 높일 계획이다.

참 고 문 헌

[1] A. Basu, V. Buch, W. Vogels, T. von Eicken, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," Proceedings of Symposium on Operating Systems Principles (SOSP), December 1995.

[2] S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," Proceedings of Supercomputing (SC), 1995.

[3] L. Prylli and B. Tourancheau, "BIP: a new protocol designed for high performance networking on myrinet," Proceedings of the International Parallel Processing Symposium Workshop on Personal Computer Based Networks of Workstations, 1998.

[4] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd, "The Virtual Interface Architecture," IEEE Micro, Vol.18, No.2, pp. 66-76, March/April 1998.

[5] J. Chase, A. Gallatin, and K. Yocum, "End-System Optimizations for High-Speed TCP," IEEE Communications, special issue on TCP Performance in Future Networking Environments, Vol.39, No.4, April 2001.

[6] H.-W. Jin, P. Balaji, C. Yoo, J.-Y. Choi, and D. K. Panda, "Exploiting NIC Architectural Support for Enhancing IP based Protocols on High Performance Networks," Journal of Parallel and Distributed Computing, Vol.65, No.11, pp. 1348-1365, November 2005.

[7] Intel Corporation, http://www.intel.com.

[8] Advanced Micro Devices, Inc., http://www.amd.com.

[9] Samsung Electronics Co., LTD, http://www. samsung.com.

[10] Tyan Computer Corporation, http://www.tyanpsc. com

[11] L. Chai, A. Hartono, and D. K. Panda, "Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters," Proceedings of The IEEE International Conference on Cluster Computing (Cluster 2006), September 2006.

[12] P. Geoffray, C. Pham, and B. Tourancheau, "A Software Suite for High-Performance Communications on Clusters of SMPs," Cluster Computing, Vol.5, No.4, pp. 353-363, October 2002.

[13] T. Takahashi, S. Sumimoto, A. Hori, H. Harada, and Y. Ishikawa, "PM2: High Performance Communication Middleware for Heterogeneous Network Environments," Proceedings of Supercomputing (SC2000), 2000.

[14] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda, LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster, Proceedings of the 2005 International Conference on Parallel Processing (ICPP-05), pp. 184-191, June 2005.

[15] Top500 Supercomputer Sites, http://www.top 500.org/.

[16] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," June 1995.

[17] L. Chai, S. Sur, H.-W. Jin, and D. K. Panda, "Analysis of Design Considerations for Optimizing Multi-Channel MPI over InfiniBand," Proceedings of Workshop on Communication Architecture for Clusters (CAC 2005), April 2005.

[18] InfiniBand Trade Association, "InfiniBand Architecture Apecification," Release 1.0, October, 2000.

[19] MPI over InfiniBand Project, http://nowlab.cse.ohio-state.edu/projects/ mpi-iba/.

진 현 욱

1997년 고려대학교 전산학(학사). 1999년 고려대학교 전산학(석사). 2003년 고려대학교 컴퓨터공학(박사). 2003년~2006년 미국 오하이오 주립대학교 연구원. 2006년~현재 건국대학교 컴퓨터공학부 조교수. 관심분야는 운영체제, 클러스터 컴퓨팅, 임베디드 컴퓨팅, 고속 네트워크

강 현 구

2001년~현재 건국대학교 컴퓨터공학부 재학중. 2008년 2월 졸업 예정

김 종 순

2001년~현재 건국대학교 컴퓨터공학부 재학중. 2008년 2월 졸업 예정