DOI: 10.3745/KIPSTA.2009.16-A.2.113

멀티코어 시스템에서 흐름 수준 병렬처리에 기반한 리눅스 TCP/IP 스택의 성능 개선

권 희 웅[†] · 정 형 진[††] · 곽 후 근[†††] · 김 영 종[††] · 정 규 식[††††]

요 약

최근 멀티코어가 장착된 시스템이 증가하면서 이를 통한 애플리케이션 성능향상에 대한 노력이 계속 되어왔다. 하나의 시스템에 다수의 처리장치가 존재함으로 인해 프로세싱 파워는 기존보다 증가했지만 기존의 소프트웨어나 하드웨어들은 싱글코어 시스템에 적합하게 설계된 경우가 많아 멀티코어의 이점을 충분히 활용하지 못하고 있는 경우가 많다. 기존의 많은 소프트웨어들은 멀티코어 상에서 공유 자원에 대한 병목현상과 비효율적인 캐시 메모리 사용으로 인하여 충분한 성능향상을 기대하기 어려우며 이러한 문제점들로 인하여 기존 소프트웨어는 코어의 개수에 비례한 성능을 얻지 못하며, 최악의 경우 오히려 감소될 수 있다.

본 논문에서는 TCP/IP를 사용하는 기존의 네트워크 애플리케이션과 운영체제에 흐름 수준 병렬처리 기법을 적용하여 성능을 증가 시킬 수 있는 방법을 제안한다. 제안된 방식은 개별 코어단위로 네트워크 애플리케이션, 운영체제의 TCP/IP 스택, 디바이스 드라이버, 네트워크 인터페이스가 서로 간섭 없이 작동할 수 있는 환경을 구성하며, L2 스위치를 통해 각 코어 단위로 트래픽을 분산하는 방법을 적용하였다. 이를 통해 각 코어 간에 애플리케이션의 데이터 및 자료구조, 소켓, 디바이스 드라이버, 네트워크 인터페이스의 공유를 최소화하여, 각 코어간의 자원을 차지하기 위한 경쟁을 최소화하고 캐시 히트율을 증가시킨다. 이를 통하여 8개의 멀티코어를 사용하였을 경우 네트워크 접속속도와 대역폭이 코어의 개수에 따라 선형적으로 증가함을 실험을 통해 입증하였다.

키워드 : 멀티코어, TCP/IP 스택, 캐시, 웹서버, 본딩, 흐름 수준 병렬처리

A Performance Improvement of Linux TCP/IP Stack based on Flow-Level Parallelism in a Multi-Core System

Huiung Kwon[†] · Hyungjin Jung[††] · Hukeun Kwak[†††] · Youngjong Kim[††] · Kyusik Chung[††††]

ABSTRACT

With increasing multicore system, much effort has been put on the performance improvement of its application. Because multicore system has multiple processing devices in one system, its processing power increases compared to the single core system. However in many cases the advantages of multicore can not be exploited fully because the existing software and hardware were designed to be suitable for single core. When the existing software runs on multicore, its performance improvement is limited by the bottleneck of sharing resources and the inefficient use of cache memory on multicore. Therefore, according as the number of core increases, it doesn't show performance improvement and shows performance drop in the worst case.

In this paper we propose a method of performance improvement of multicore system by applying Flow-Level Parallelism to the existing TCP/IP network application and operating system. The proposed method sets up the execution environment so that each core unit operates independently as much as possible in network application, TCP/IP stack on operating system, device driver, and network interface. Moreover it distributes network traffics to each core unit through L2 switch. The proposed method allows to minimize the sharing of application data, data structure, socket, device driver, and network interface between each core. Also it allows to minimize the competition among cores to take resources and increase the hit ratio of cache. We implemented the proposed methods with 8 core system and performed experiment. Experimental results show that network access speed and bandwidth increase linearly according to the number of core.

Keywords : Multicore, TCP/IP Stack, Cache, Webserver, Bonding, Flow-Level Parallelism

※ 본 연구는 숭실대학교 교내 연구비 지원으로 이루어졌음.
† 정 회 원 : 숭실대학교 정보통신전자공학부 박사과정
†† 정 회 원 : 숭실대학교 정보통신전자공학부 석사과정
††† 정 회 원 : 숭실대학교 정보통신전자공학부 박사(교신저자)
†††† 정 회 원 : 숭실대학교 정보통신전자공학부 교수
논문접수 : 2009년 2월 25일
수 정 일 : 1차 2009년 3월 23일
심사완료 : 2009년 3월 23일

1. 서 론

멀티코어 성능의 문제점들에 대한 내용은 저수준의 프로그래머들이나 하드웨어 설계자들 중에도 일부에게만 익숙한 내용들이다. 일반적으로 고수준 언어를 사용하거나 서버를 운영하는 측면에서는 멀티코어에 대한 성능향상 기법을 모두 파악하여 프로그래밍하거나 시스템을 운영하는 것은 쉽지 않은 일이라 할 수 있다. 본 논문과 향후 연구는 멀티코어 사용자들에게 좀 더 쉬운 성능향상 기법을 운영체제나 하드웨어 측면에서 제공하는 것을 목표로 하고 있다.

최근 중앙 처리 장치(CPU)의 성능은 클럭(clock)을 높이는 방법에서, 코어의 수를 늘리는 방향으로 변화되고 있다. 코어 수의 증가는 전체적인 처리용량의 증가를 의미하지만, 이는 소프트웨어의 구조도 이에 적합하게 고안되어야 한다는 점을 의미한다. 범용 서버나 컴퓨터(PC)에서 주로 사용하는 구조인 대칭형 다중 처리(SMP: Symmetric Multi Processing) 시스템은 다수의 코어가 메인메모리를 공유함으로서 발생하는 문제점으로 인하여 코어 수에 비례하여 소프트웨어 성능이 쉽게 증가하지 않는 문제점이 존재한다.

이러한 요인의 대표적인 예는 다음과 같다. 운영체제 내에서 다수의 코어가 접근해야 하는 공용 자료구조를 보호하기 위해 락(lock) 기법을 통해 한 번에 한 개의 코어만이 이를 접근하도록 허용하는데, 멀티코어 시스템에서는 이러한 경우 코드가 병렬로 처리되지 못하고, 직렬로 처리가 되는 현상이 발생하여 성능이 저하된다. 또한, 한 코어에서 이러한 자료구조의 내용을 읽는 경우 캐시메모리에 저장되고, 이를 변경한 경우, 변경된 내용이 캐시 메모리에 저장되는데, 이를 다른 코어에서 읽거나 변경을 시도 하는 경우 변경된 내용이 메인메모리에 먼저 저장된 후, 해당 코어의 캐시 메모리로 적재가 되어야 한다. 이러한 자료구조가 여러 개의 코어에 의해서 빈번하게 접근되면 각 코어간의 캐시 메모리에 핑퐁(ping-pong) 현상처럼 메인메모리를 통해 자료구조의 내용들이 이동하는 캐시 바운싱(Bouncing) 현상이 발생하며, 이때에는 캐시 히트율이 급감하며, 느린 메인메모리를 통한 데이터의 이동으로 인하여 성능이 현저하게 떨어지는 현상을 보인다.

따라서 소프트웨어 설계 시에 이를 감안하여 각각의 코어가 메모리를 분할하여 독립적으로 연산하고, 최종 결과를 합산하는 식과 같은 방법을 적용하여 가급적 코어 간 공유 메모리 영역을 차지하기 위한 경쟁을 피하고, 캐시히트(cache hit)율을 높여 성능을 증가 시킬 수 있다[1].

일반적으로 운영되는 네트워크 애플리케이션을 이와 같은 관점에서 살펴보면 다음과 같은 문제를 가진다고 볼 수 있다. 일예로 웹서버를 살펴보면 한 개의 네트워크 인터페이스와 한 개의 리스닝(listening) 소켓, 그리고 다수의 프로세스를 가지는 웹 데몬이 멀티코어 시스템에서 동작한다. 기본적으로 다수의 프로세스는 여러 개의 코어에 분산되어 작동을 하며, 이들이 리스닝 소켓과 TCP/IP 스택내의 여러 가지 자료구조를 공유하게 된다. 또한, 각 코어에서 웹 서버 프로세스/쓰레드를 통해 송수신되는 데이터 들은 TCP/IP 스택을 거쳐 네트워크 인터페이스를 통해 외부와 통신이 이루어지는데, 이 역시도 유일한 공유자원에 대한 접근이라고 볼 수 있다. 또한, 네트워크를 활용하는 애플리케이션 또한 소켓이나 네트워크 인터페이스 외에 자료구조나 데이터의 저장소인 파일, 데이터베이스 등의 공유되는 리소스를 활용하기 때문에 성능 저하가 발생할 수 있다.

앞서 설명한 바대로 각 코어가 이러한 공유자원을 차지하기 위한 경쟁을 하게 되며, 각 단계의 코드 수행에 있어서 공유 자원을 차지하기 위한 경쟁 현상이 발생되어 성능이 코어의 수에 비례하여 증가하지 않거나, 최악의 경우 전체적인 직렬화로 인하여 성능이 현저하게 저하되는 현상이 나타날 수 있다.

본 논문에서는 앞서 설명한 문제점을 해결하는 방법을 제시하는데, 여기서는 가급적 애플리케이션 코드의 수정을 최소화 하는 방향에서 접근을 시도하며, 네트워크를 통한 패킷 송수신과 관련된 소프트웨어적인 문제와 하드웨어적인 문제, 그리고 TCP/IP 스택에서의 공유 자원을 최소화 할 수 있는 방법을 제안하며, 또한 기존에 만들어진 애플리케이션을 가급적 수정 없이 활용할 수 있는 방법인 흐름 수준 병렬처리 기법을 적용하였다. 이는 패킷의 플로우 즉, TCP의 경우 한 개의 커넥션에 대한 모든 패킷 처리와 애플리케이션에 대한 처리가 한 개의 코어에서 실행될 때, 높은 성능을 나타낼 수 있음을 의미한다.

제안하는 방법은 패킷을 수신한 네트워크 인터페이스와 패킷을 처리하는 애플리케이션이 동일한 코어에서 작동하도록 하여, 네트워킹 커널부와 애플리케이션이 코어 단위로 가급적 독립적인 메모리 영역을 사용하면서 작동하도록 하였다. 이를 위해서 코어의 개수와 동일한 수의 네트워크 인터페이스를 사용하였으며, 각 네트워크 인터페이스에는 L2 스위치를 이용하여 패킷을 플로우 단위로 분산 처리하는 방법을 사용하였다.

본 논문의 구성은 다음과 같다. 2장에서는 기존 연구를 소개하고, 3장에서는 흐름 수준 병렬처리에 기반한 제안된 방법을 소개한다. 4장에서는 실험 및 결과를 소개하고 마지막으로 5장에서는 결론 및 향후 연구 방향으로 끝을 맺는다.

2. 기존의 흐름 수준 병렬처리

2.1 네트워크 상에서의 흐름 수준 병렬처리

리눅스 운영체제(O/S)에서 멀티코어에 대한 성능 실험과, 다양한 측면에서의 개선시도가 진행되어 왔다. 특히 네트워크 관점에서는 흐름 수준 병렬처리에 대한 연구가 많이 진행되었는데, 다수의 코어가 동일한 소프트웨어가 작동하는 멀티코어 환경에서 서로 다른 패킷 플로우, 즉 TCP의 경우 커넥션 단위로 패킷 송수신부터 TCP/IP 스택, 그리고 애플리케이션까지 한 개의 코어에서 처리되는 것을 의미한다. 이를 통해, 코어 단위로 서로 커널 내부의 자료 구조에 대한 독립적 사용을 가능하게 하여 캐시 히트율과 공유 메모리 영역의 접근으로 인한 코드 수행 직렬화 문제를 최소화

하였을 때 높은 성능을 얻을 수 있다고 가정하고 있다.

흐름 수준 병렬처리에 대한 예로 인텔에서 오픈소스 침입탐지시스템인 스노트(snort)를 멀티코어에 적합하게 수정하여 선형적인 성능을 이끌어 낸 예가 있다[2,3]. 4개의 코어를 장착한 시스템에서 한 개의 코어는 패킷을 플로우 단위로 분할하여 나머지 3개의 코어에 부하 분산하여 전달하면 나머지 3개의 코어에서 작동하는 스노트 프로세스들은 이들에 대해 패턴을 이용해서 악의적인 트래픽인지를 판단한다. 이러한 구조를 적용하였을 때, 기존 방식을 그대로 적용하였을 때보다 세션 처리에서 약 6배 정도의 성능 개선효과를 얻을 수 있었다. 이는 패킷 플로우들이 각각의 코어로 나누어 처리함에 따라서 각각의 코어에서 처리하는 세션수가 적어지게 되어 캐시 히트율이 증가하고, 또한 각 코어들이 세션정보를 공유하지 않게 되어 이를 액세스하기 위한 경쟁이 최소화되기 때문이다. 이 실험은 흐름 수준 병렬처리의 효과를 입증해 주고 있다. 이 실험은 기존의 TCP/IP 스택을 사용하지 않고 네트워크 인터페이스에서 수신된 패킷을 곧바로 처리하기 때문에 쉽게 접근할 수 있는 방법이다.

네트워크 인터페이스에 대한 연구로는 네트워크 인터페이스에서 패킷을 플로우(Flow)단위로 분할하여 각각의 코어에서 나누어 처리하는 기법에 대한 연구가 있었다[4]. 이 경우에는 TCP/IP 스택이나 애플리케이션은 기존의 시스템을 그대로 사용하였으며, 미리넷(Myrinet)의 프로그래밍 가능한 네트워크 인터페이스의 펌웨어를 수정하여 수신한 패킷을 플로우 단위로 분할하여 각각의 코어에 분산하여 전달할 수 있도록 하였으며, 디바이스 드라이버는 쓰레드를 통하여 패킷 플로우를 TCP/IP 스택에 전달하여 애플리케이션으로 전달 하 수 있도록 하였다. 이와 같은 방법으로 4개의 중앙처리 장치를 사용하였을 때 전송률을 기존 시스템에 비해서 약 30%-50% 정도 개선시킬 수 있었다.

대칭형 다중 처리상에서 닉(NIC: Network Interface Card)과 패킷수신 방식의 변경만으로 TCP/IP 스택의 작동에 영향을 준다는데 큰 의의를 가지지만 패킷의 송수신에 대해서만 플로우가 유지가 되며, 그 이상의 단계에서는 패킷의 처리에 있어 여러 개의 코어에서 처리가 되어 성능이 크게 증가하지는 못한 것으로 보인다.

또한 최근 이와 유사한 방법으로 인텔 네트워크 인터페이스의 리눅스 드라이버인 e1000 에 네트워크 인터페이스의 물리적인 송신 큐(Queue)를 2개 또는 4개까지 늘려 성능을 개선하려는 시도들이 일어나고 있다[5]. 하지만 패킷의 송신에 대해서만 이러한 변경사항이 반영되었을 뿐이며, 수신시에는 한 개의 코어만이 처리할 수 있다는 기존의 문제점을 그대로 가지고 있다. 또한 코어의 수가 4개 이상일 경우에는 기존의 방식과 같이 해당 큐들이 공유되는 문제점이 발생하는 단점이 있어 아직까지 근본적인 해결책을 제시하고 있지는 못하고 있다. 이상과 같은 문제점으로 인하여 흐름 수준 병렬처리의 현실적인 적용은 현시점까지는 쉽지 않다고 볼 수 있다.

다른 방향의 접근으로는 코어 단위로 별도의 TCP 스택을 사용하도록 하는 방법들이 연구되었다[6,7]. 이 방법은 각각의 코어에 TCP 스택과 애플리케이션이 별도로 작동하도록 구성되어 캐시의 활용도를 극대화 하여 높은 성능을 얻을 수 있었다. 하지만 코어단위의 개별적인 TCP 스택의 사용으로 인하여 응용프로그램 프로그래밍 인터페이스(API: Application Programming Interface)와 애플리케이션의 구조를 크게 변경해야 하는 단점이 존재하여 현실적인 사용은 어렵다고 볼 수 있다.

최근에 이보다 진보적인 연구로 최근 멀티코어 상에서 운영되는 리눅스 시스템에서 아파치 웹서버 성능을 파악하려는 시도가 있었다[8]. 이 연구에서는 실제 환경에서와 같이 웹 애플리케이션과 연계되어 작동하는 아파치 코어의 수에 따른 확장성(Scalability)이 좋지 않다는 가정에서 시작하였으며, 결론은 코어들 간의 캐시 메커니즘 문제로 인하여 더 이상 성능을 증가시킬 수 없으나, TCP/IP 스택은 코어 수에 비례하는 성능향상을 얻을 수 있다고 언급하고 있다. 하지만 TCP/IP 스택의 확장성을 입증하는 과정이 고정된 수의 커넥션을 통해 대용량에 대한 전송량 측정 테스트이기 때문에, 웹서버가 실제로 운영되는 환경에서 발생하는 빈번한 접속과 연계하여 발생하는 성능 저하 현상은 설명하지 못하고 있다. 또한 시스템 관점에서는 네트워크 인터페이스와 코어의 관계를 IRQ 유사성(Affinity)을 통하여 플로우를 보장하게 하였으나, 각각의 네트워크 인터페이스에 IP를 할당하여, 실질적인 사용보다는 성능 향상의 가능성의 검토차원에서 진행된 테스트라고 볼 수 있으며, 전체적인 시스템 운용 관점에서의 접근이 미약하다고 볼 수 있다.

앞서 언급한 연구들을 종합적으로 정리해보면, 다양한 관점에서 흐름 수준 병렬처리에 대한 연구들을 진행 중이라는 것을 알 수 있다. 하지만 TCP/IP 스택 자체에 대한 근본적인 문제에 대한 지적이나 개선 방향을 제시하지는 못하고 있으며, 현실적으로 현재 사용중인 애플리케이션을 그대로 사용하여 성능향상을 얻기에는 어려운 문제들도 존재한다. 또한, 패킷 송수신부터 애플리케이션의 처리까지 전체적인 부분을 포괄하는 접근 방법은 아직까지는 부족하다고 볼 수 있으며, 현재까지는 부분적인 접근이 많았다고 판단할 수 있다.

2.2 리눅스 시스템에서의 멀티코어 지원

리눅스 시스템에서는 각종 디바이스들에 대해 IRQ 유사성[9]이라는 메커니즘을 통해 각 디바이스의 인터럽트를 처리하는 코어를 한 개 또는 다수를 지정할 수 있다. 한 개를 지정한 경우는 특정 인터럽트가 지정된 특정코어에서만 처리한다. 다수의 코어를 지정한 경우는 커널 내부의 irqbalance라는 커널쓰레드가 지정된 코어들의 인터럽트 빈도를 주기적으로 모니터링 하여, 가장 부하가 적은 코어에서 특정 디바이스의 인터럽트가 처리 될 수 있도록 재지정 하는 과정을 반복하여 시스템의 부하가 최소화 되도록 한다.

하지만 일반적으로 서버 애플리케이션을 운영하는 경우 네트워크 입출력(I/O)이나 디스크 입출력과 같이 빈번하게 사용되는 디바이스들의 경우 IRQ를 고정하여 특정 코어만

이 이들을 처리할 수 있도록 지정하는 경우 더 높은 성능을 내는 경우가 많다. 네트워크 입출력의 경우 IRQ 유사성 적용을 통해 듀얼코어 시스템에서 약 25%의 성능향상이 있었다. 대부분의 성능향상은 메모리와 연관된 부분에 이었으며 캐시미스의 감소와 성능 향상치가 부합된다는 실험 결과를 얻을 수 있었으며 IRQ 유사성을 지정하는 경우 캐시의 효율성을 향상시킨 다는걸 알 수 있다[10].

또한 운영체제인 리눅스 시스템 또한 최근 몇 년 사이에 대칭형 다중 처리/멀티코어 시스템 지원을 위한 다양한 기법들이 적용되고 있다. 그 중 네트워킹 시스템과 관련한 부분을 살펴보면 다음과 같다.

리눅스 커널 내의 여러 가지 일들을 수행하는 쓰레드들이 기존에 한 개에서 최근에는 코어의 수만큼으로 늘어났다. 그 중 대표적인 것은 ksoftirq를 통한 패킷 처리이다. 이를 통해 디바이스 드라이버를 통해 수신하는 패킷들을 처리하기 위해 코어의 개수에 맞추어 자료구조를 보유하고 있으며, 각각의 코어들이 패킷의 송수신시 독립적으로 작동할 수 있는 기반을 가지고 있다[11].

또한 네트워크 시스템의 라우팅 엔트리 들이나 관련 자료구조들을 다수의 코어가 동시에 빠르게 접근할 수 있도록 하기 위하여 기존의 읽기/쓰기(read/write) 락(lock)이나 spinlock 대신 기존의 락(lock)과 달리 한 개의 링크드 리스트(Linked List) 내에 있는 서로 다른 데이터를 동시에 읽고 쓸 수 있는 RCU(read-copy update)[12, 13]를 통하여 보호하기 때문에, 기존에 비해 코드 직렬화 문제를 현저히 줄일 수 있으며 주로 네트워크 서브시스템에 주로 적용되었다.

이상과 같은 방법들은 현재 리눅스의 네트워크 시스템이 어느 정도 멀티코어 시스템에 적합하도록 변해가고 있다는 점을 시사하고 있다. 하지만 이 또한 마찬가지로 전체적인 변화가 한꺼번에 이루어지는 것은 아니고, 주로 네트워크 시스템의 라우팅과 관련한 부분에서 많은 변화가 일어났으며, 이러한 개선사항들이 점차 확대되어 나갈 것으로 보인다.

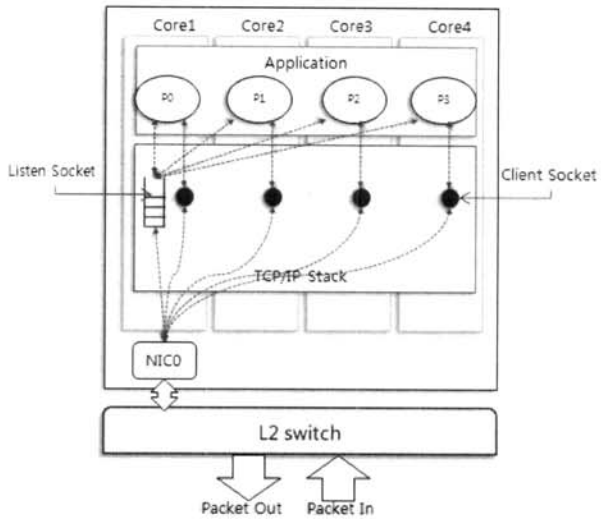
3. 제안된 흐름 수준 병렬처리

3.1 일반적으로 운영하는 네트워크 서버모델

일반적으로 멀티코어를 사용하는 시스템에서의 웹서버는 (그림 1)과 같은 형태를 가진다. 한 개의 네트워크 인터페이스에 한 개의 IP 주소가 할당되어 있으며, 여기에 웹 서버 애플리케이션이 한 개의 리스닝 소켓을 통하여 외부의 접속을 수신하며, 다수의 웹 서버 프로세스들에게 접속들을 배분하여 처리하는 구조를 가진다. 물론, 다수의 IP 주소와 리스닝 소켓을 가지는 형태로 운영할 수도 있으나, 한 개의 서비스 관점에서 보면 앞서 설명한 바와 같다고 볼 수 있다.

이러한 시스템의 운영에 있어 다수의 사용자의 접속이 발생할 경우 네트워크 입출력 관점에서의 성능상의 문제점을 살펴보면 다음과 같은 사항들로 인하여 멀티코어를 사용하는데 있어 충분한 성능향상을 기대하기 어렵다.

네트워크 인터페이스는 특정 코어에 IRQ가 할당되어 있



(그림 1) 기존 시스템에서의 웹서버 운영

어, 다수의 코어 중 한 개의 코어만이 실질적인 외부와의 패킷의 송수신을 담당하여 부하가 집중된다고 볼 수 있으며, 시스템 성능의 병목을 가지고 올 수 있는 부분이다. 일차적으로 작업 처리량(throughput)에 대한 병목 지점이 된다.

패킷 송수신을 통해 사용자의 접속을 처리하는 과정에서는 요청된 사용자의 접속은 리스닝 소켓의 접속 큐에 그 내용이 기록되고, 각 코어에서 작동하는 애플리케이션 프로세스들이 경쟁적으로 접속 큐를 통하여 사용자의 접속을 획득한다. 리스닝 소켓과 접속 큐는 각 코어 상에서 작동하는 애플리케이션 프로세스들 관점에서 공유되는 자료구조로 락(lock)을 통해 보호되며, 접속을 대기하고 있는 프로세스들이 이를 획득하기 위한 경쟁을 하는 과정에서 코드 직렬화와 캐시미스 등이 빈번하게 발생하여 성능에 좋지 않은 영향을 끼치며, 이 결과로 접속 속도가 낮아지는 문제가 발생한다. 이는 접속이 빈번하게 일어나는 웹 서버와 같은 애플리케이션에서는 응답성이 상당히 떨어지는 문제점을 가질 수 있다.

접속이 이루어진 이후에는 시스템 내부에 사용자 소켓(Client Socket)이 생성되어 요청자와 애플리케이션 프로세스간의 통신이 이루어지는데, 이 또한 사용자 소켓이 패킷 수신시에는 네트워크 인터페이스와 연결된 코어에서 처리가 되고, 애플리케이션 처리 시에는 다른 코어에서 처리되는 불일치를 보인다. 이 과정에서도 사용자 소켓은 패킷을 수신하는 코어와 이를 처리하는 코어가 일치하지 않아 마찬가지의 코드 직렬화와 캐시미스가 발생하며, 작업 처리량을 저하시키는 요인이 된다.

마지막으로 각각의 코어에서 애플리케이션에서 처리된 패킷이 네트워크상으로 전송되는 과정에서는 각 코어상의 애플리케이션 프로세스들이 패킷을 송신하기 위해 네트워크 인터페이스와 연결된 패킷 송신 큐에 해당 패킷을 등록하여야 한다. 이러한 송신 큐 네트워크 인터페이스 단위로 한 개의 송신 큐를 가지고 있으며, 락(lock)으로 보호되고 있어 다수의 코어에서 패킷 전송을 시도하는 경우 코드 직렬화와

캐시미스가 발생하여 성능저하가 발생하고 마찬가지로 작업 처리량을 저하시키는 요인이 된다.

현 시스템에서 다수의 네트워크 인터페이스와 여기에 각각의 IP 주소를 할당하여 성능을 증가시키는 방법을 생각해 볼 수 있으나, 이 또한 근본적인 성능저하 요인을 해결하지 못한다. 다시 말하면 각각의 네트워크 인터페이스를 IRQ를 통하여 각각의 코어와 연결하고, 이에 대한 별도의 IP 주소를 할당하여 애플리케이션을 구동시키는 경우, 각각의 네트워크 인터페이스와 IP 주소 대해서 처리되는 과정을 살펴보면 앞서 설명한 문제처럼 특정 IP 주소를 통해 네트워크로 송수신되는 패킷들은 한 개의 코어에서만 처리가 되며, 이러한 패킷들은 다수의 애플리케이션 프로세스들에게 분산되어 처리된다는 것을 유추할 수 있다. 따라서 네트워크 인터페이스를 늘이면서 약간의 성능향상이 있을 수 있지만, 시스템 내부에서의 락(lock)을 통한 코드 직렬화나 캐시미스 등의 영향으로 인한 성능저하는 문제는 해결되지 않고 그대로 남아 있다고 판단할 수 있다.

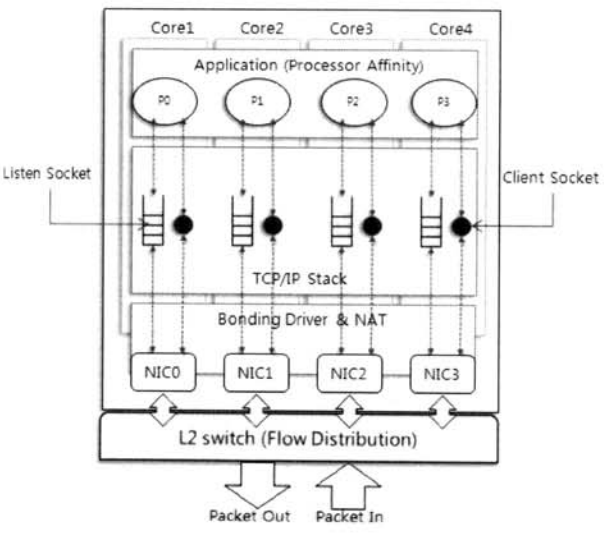
결론적으로 일반적으로 운영되는 시스템에서는 다음과 같은 문제점을 가지고 있다고 볼 수 있다. 애플리케이션에서 처리되는 패킷들의 송수신 과정에 있어서, 네트워크 인터페이스, 리스닝 소켓, 그리고 사용자 소켓 등이 둘 또는 그 이상의 코어들 간에 공유가 되고, 운영체제 내에서 이들을 처리하는 자료구조들이 락(lock)으로 보호됨에 따라, 코드의 수행이 직렬화 되거나 자료구조에 대한 접근 시에 캐시 바운싱 현상이 빈번하게 발생하여 성능저하를 유발한다. 이러한 상황에서 네트워크 입출력이 많은 부분을 차지하는 애플리케이션의 경우 현격한 성능저하가 일어날 수 있으며, 앞서 언급한대로 코어의 수를 증가시키더라도 이에 비례하는 성능향상을 얻기는 어렵다.

3.2 흐름 수준 병렬처리의 적용

문제를 개선함에 앞서 일반적인 서버의 운영에 대한 가정을 두었으며 최대한 사용자 관점에서는 일반적인 모델과 유사한 환경을 가질 수 있도록 하였다. 따라서 제안하는 시스템도 외형적으로는 한 개의 IP 주소와 한 개의 외부 네트워크 인터페이스를 가지는 형태로 구성하였다. 이렇게 하는 이유는 단일 또는 다수의 시스템 운영 시에 IP 주소의 낭비 문제 그리고 관리 요소를 줄일 수 있으며, 또한 다수의 IP를 사용할 외부에서 사용자들이 접속하는 주소를 통일하기 위해서 L4 스위치 또는 도메인 이름 서비스(DNS: Domain Name System) 등을 이용한 부하분산 방법 등을 통해 이러한 문제를 해결해야 하는데, 비용적인 문제나 운영의 복잡성 문제가 존재하기 때문에 이러한 사항을 가급적 시스템 내부에서 처리 하도록 하였다. 또한, 본 시스템에서는 다량의 커넥션이 발생하는 웹서버의 운영환경을 가정하고 있으며, 소수의 사용자가 많은 대용량의 트래픽을 유발하는 상황에서는 트래픽이 특정코어에 집중되어 처리되는 현상은 고려하지 하지 않는다. 앞서 설명한 도메인 이름 서비스 또는 L4 스위치를 이용한 부하분산 방식을 예로 들면, 다수의 서버가 트래픽을 처리하지만, 한 개의 커넥션은 한 개의 서버로 할당되며, 한 개의 커넥션에 대한 최대 속도는 한 개의 서버에서 낼 수 있는 최고 속도가 된다. 또한, 한 대의 서버내에서도 한 개의 대용량 커넥션에 대한 애플리케이션의 처리가 특정코어로 한정되기 때문에 한 개의 코어의 성능에 의존적이라고 볼 수 있다. 본 연구에서도 한 개의 커넥션에 대해서 한 개의 코어가 처리한다는 가정을 하고 있음으로 커넥션 한 개의 처리 성능은 한 개의 코어에 의존적이라고 가정한다.

제안하는 시스템은 기본적으로 여러 개의 코어가 한 개의 패킷 플로우를 다수의 코어에서 분할하여 처리하는 것을 금지하여 성능에 대한 향상을 시도하였다. 구체적으로는 앞서 문제점으로 언급하였던 리스닝 소켓, 사용자 소켓, 그리고 네트워크 인터페이스가 두 개 이상의 코어에서 접근되는 것을 회피할 수 있도록 각각의 코어가 대부분의 시스템 리소스를 독립적으로 가질 수 있는 방향으로 시스템이 설계되었으며, 각각의 코어는 자신만의 네트워크 인터페이스와 리스닝 소켓, 그리고 사용자 소켓과 이를 처리하는 애플리케이션 데몬을 가지며 이들은 다른 코어들과 공유되지 않도록 설계 하였다.

(그림 2)에서와 같이 각각의 네트워크 인터페이스는 자신과 대응하는 코어와 IRQ 유사성을 통해서 연결된다. 그리고 시스템내의 모든 네트워크 인터페이스들은 L2 스위치의 링크 수집(Link Aggregation)과 본딩 드라이버(bonding driver)[14]를 통하여 다수의 네트워크 인터페이스를 한 개의 가상 인터페이스로 통합을 하며, 멀티코어 시스템 내부에서는 한 개의 네트워크 인터페이스를 통해 외부와 패킷의 송수신을 하는 것처럼 작동한다. 그리고 L2 스위치는 링크 수집을 적용한 포트들에 대하여 부하분산 알고리즘을 지원하는데, 제안된 시스템에서는 외부에서 유입되는 패킷들에 대하여 출발지와 목적지 주소를 해싱(hashing)[15]하여 패킷이 플로우 단위로 섞이지 않고 각각의 코어와 연결된 네트워크 인터페이스로 전달 될 수 있도록 하여 패킷의 송수신



(그림 2) 제안된 시스템의 구조

시에 있어서 패킷 플로우와 이들을 처리하는 코어를 일치되도록 하였다. L2 스위치의 부하분산을 사용하는 경우 소수의 커넥션이 대량 트래픽을 유발하는 경우나, 의도적으로 특정 패턴의 IP 주소를 가지는 트래픽이 발생할 경우 특정 코어에 부하가 집중될 수 있는 문제점이 존재하나, 웹서버와 같이 많은 수의 커넥션이 발생하는 경우에는 코어에 평균적으로 커넥션이 분산되어 이러한 문제점이 거의 발생하지 않는다고 가정하였다.

패킷을 수신하는 코어와 이를 처리하는 프로세스들과의 플로우 유지를 위해서는 각각의 코어마다 프로세스(process) 유사성[16]을 통해 동일한 애플리케이션 데몬을 하나씩 할당하며, 파생되는 프로세스 또한 동일한 코어에서 작동하도록 하였다. 그리고 각 데몬의 리스닝 소켓은 서로 다른 TCP 포트를 할당하여 사용자의 요청을 받아들이도록 하였다. 이를 통하여 각각의 코어는 동일 애플리케이션에 대해서 별도의 리스닝 소켓을 가지게 된다. 이들 리스닝 소켓은 특정 코어에 할당 할 수는 없으나, 해당 자료구조를 참조하는 코어가 한 개로 제한되기 때문에 앞서 설명한 캐시 바운싱 현상이나 락(lock)을 통한 코어들의 코드 직렬화 수행 문제는 현저하게 줄어들어 특정 코어에 IP 주소를 할당한 것과 동일한 결과를 얻어 낼 수 있다.

그리고 시스템 외부에서 사용자들이 접속하는 TCP 포트와 시스템 내부의 각 코어에서 작동하는 애플리케이션 데몬의 TCP 포트가 다르기 때문에 패킷이 시스템으로 유입되는 경우 목적지 TCP 포트를 각 코어에 지정된 TCP 포트로 변경하고, 외부로 패킷을 송신하는 경우에 이를 원래 TCP 포트로 복원할 수 있는 네트워크 주소 변환(NAT: Network Address Translation)를 추가하여, 내외부로 패킷이 전달되는 경우에 대한 문제가 없도록 하였다.

이상과 같이 다소 복잡한 방법을 통하여 애플리케이션을 수행하는 것 보다는 커널 내에서 다수의 리스닝 소켓을 생성하거나, 코어단위로 리스닝 소켓의 자료구조를 분리하는 방법을 통해 기존의 수많은 애플리케이션들을 수정하지 않고, 일반적인 방법을 통해 사용하는 이상적인 형태이나, 현 시점에서는 제안된 방식이 성능향상을 얻을 수 있다면 앞서 설명한 방법 또한 성능이 향상된다는 가정을 두고 진행하였다.

이 시스템을 구현하는 과정에서 리눅스에서 제공하는 본딩 드라이버는 멀티코어에서 상당한 성능저하가 나타나는 것을 발견하였다. 본딩 드라이버는 자신이 가상화한 다수의 네트워크 인터페이스들에 대해서 라운드로빈(Round-Robin)이나 트래픽에 기반한 부하분산 알고리즘을 통해서 네트워크 인터페이스를 지정하여 패킷을 송신한다. 이때 앞서 설명한 바와 같이 패킷을 송신하려고 하는 코어와 실제로 송신하는 코어가 다른 경우 네트워크 인터페이스에 대한 락(lock)과 캐시미스 등이 심각하게 발생하였다. 따라서 제안된 시스템에서는 본딩 드라이버에서 제공하는 부하분산 알고리즘을 사용하지 않고, 패킷 전송을 시도하는 코어에 IRQ 유사성으로 연결된 네트워크 인터페이스를 통해서만 패킷을 송신하는 구조로 변경하여 성능 저하에 대한 문제를 해결하였다.

이상과 같은 시스템에서의 패킷의 흐름을 살펴보면 L2 스위치를 통해 유입된 패킷들은 링크 수집 부하분산 알고리즘을 통하여 패킷을 전송할 포트가 선정되며, 선택된 네트워크 인터페이스를 통해 수신된 패킷은 IRQ 유사성으로 연결된 코어에 의해서 처리가 된다. 이렇게 수신된 패킷은 해당 코어에 지정된 TCP 포트로 네트워크 주소 변환이 되어 TCP/IP 스택으로 전달된다. 이들 패킷은 TCP/IP 스택으로 전달되어 접속에 대한 처리, 애플리케이션 프로세스들의 처리, 패킷 송수신에 대한 모든 처리가 한 개의 코어 상에서 이루어진다. 따라서 앞서 설명한 코어들 간의 공유되는 자료구조의 사용으로 인한 캐시 미스와 락(lock)으로 인한 코어간의 코드 수행 직렬화 문제가 상당 부분 해결되어 코어의 수에 따른 확장성을 기대할 수 있다. 그리고, 한 개의 리스닝 소켓에 대한 커넥션 수용 능력은 이론적으로 TCP 스택의 스펙대로 포트 레인지(최대 65535개)를 넘어설 수 없지만, 네트워크 주소 변환을 통해 각 코어상의 애플리케이션이 각각의 리스닝 소켓를 가지는 구조를 통해 코어 개수의 배수만큼의 커넥션 수용능력을 가지는 부수적인 효과를 얻을 수 있다.

4. 실험 및 토론

4.1 실험 환경

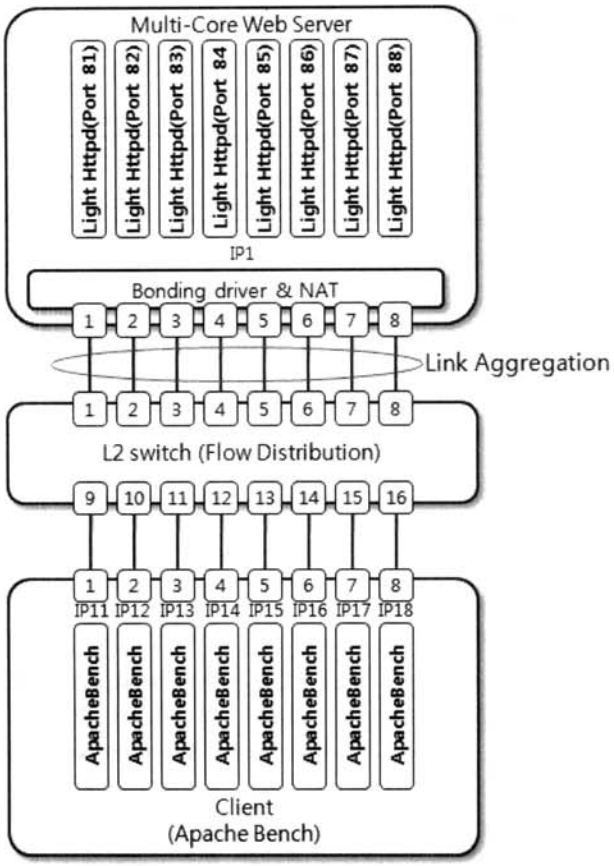
실험은 일반적인 서버운영 모델과 제안된 방법의 접속속도 측정을 통한 확장성을 비교하는 방향으로 진행하였다. 확장성을 측정하기 위해서 리눅스 시스템에서 코어의 수를 늘려가면서 성능의 변화를 측정하였다. 또한 앞서 언급했던 성능 저하 요인들을 확인하기 위해서 성능 측정 시에 프로파일러(Profiler)를 이용하여 커널 내부에 성능 문제가 발생하는 위치들을 추적하였다.

실험에 사용한 시스템의 사양은 <표 1>과 같으며 코어가 4개 탑재된 인텔 제온 중앙 처리 장치 및 리눅스 커널의 부팅 파라미터를 이용하여 작동하는 코어의 수를 1개에서부터 8개까지 늘려가면서 실험을 진행하였다.

실험구성은 (그림 3)과 같으며, 제안된 시스템을 위하여 8개의 네트워크 인터페이스를 L2 스위치와 링크 수집을 통해서 연결하였고 테스트용 클라이언트를 위해서 8개의 인터페이스를 L2 스위치와 연결하였으며, 본딩 드라이버로 생성된 가상 인터페이스에 웹서버의 대표 IP 주소를 설정하였다.

〈표 1〉 테스트 서버 및 클라이언트 시스템 사양

서버/클라이언트	중앙 처리 장치	2개의 인텔 제온 하퍼타운 2.5Ghz(E5420)
	메모리(RAM)	2G bytes
	닉(NIC)	8개의 intel 1G NIC(PCIe 4배속으로 연결)
	운영체제(커널)	Fedora 7(커널 Linux 2.6.20.21)
	웹서버	Lighttpd 1.4.20
L2 스위치	모델	다산 V5224G
	포트	24 Port 10/100/1000 Base-TX
	기타	링크 수집 지원



(그림 3) 테스트 환경

제안된 시스템에 탑재된 웹서버는 실험을 용이하게 할 수 있도록 단일 쓰레드 방식의 웹서버인 lighthttpd[17]를 사용하였다. 실험을 위해 lighthttpd가 지정된 코어에서 작동할 수 있도록 sched_set_affinity[16]라는 함수를 lighthttpd의 프로세스와 쓰레드가 생성 시에 호출 하여 수행되는 코어를 강제적으로 지정할 수 있도록 하였다. 또한 lightttpd를 개별적으로 측정하여 한 개의 코어에 2개의 쓰레드가 작동할 경우 가장 좋은 성능이 나온다는 것을 확인하였다. 실제 실험에서는 각각의 코어 단위로 한 개의 lighthttpd에 두 개의 쓰레드와 별도의 TCP 포트를 지정하여 실험을 수행하였다.

테스트 클라이언트에는 8개의 네트워크 인터페이스에 각각의 IP주소를 지정하였다. 아파치벤치[18]는 출발지 주소를 지정하는 기능을 별도로 추가하여, 테스트 시에는 각각의 포트의 IP 주소를 출발지로 지정한 8개의 아파치벤치가 동시에 수행하여 나온 결과를 집계하였다.

또한 기존 시스템에서의 성능저하 현상과 이를 개선한 제안된 방식과의 실제 동작상에서의 차이점과 개선 점들을 명확하기 위해 리눅스에서 작동하는 성능 분석용 프로파일러인 인텔의 vtune[19]을 이용하여 인스트럭션의 수행시간이 측정하는 CPI(Cycle Per Instruction)값과 캐시미스 값을 함수 단위로 측정하여 리눅스 커널 코드에서의 문제점을 분석하였다.

4.2 본딩 드라이버에 대한 실험 및 분석

개선된 본딩 드라이버의 성능의 차이를 실험을 통하여 비교하였다. 실험환경은 기존 본딩 드라이버와 개선된 본딩 드라이버 모두 4개의 인터페이스를 가상화하였다. 그리고 기존 방식에는 가장 많이 사용하는 부하분산 알고리즘인 라운드로빈을 적용하였다. 그 외의 알고리즘은 부하를 배분하는 방식에는 차이가 있으나, 네트워크 인터페이스를 선택한 후 패킷을 송신하는 문제에 있어서는 동일하고 성능상의 차이는 없다고 할 수 있기 때문에 실험 결과에 포함시키지는 않았다.

실험은 아파치벤치를 통하여 제안된 시스템내의 웹서버의 처리 요청수를 비교 측정하였으며 결과는 다음과 같다. 먼저 기존 본딩 드라이버와 개선된 본딩 드라이버의 차이점을 확실히 하기 위해 vtune을 통하여 프로파일링을 진행하였으며 그 결과는 <표 2>와 <표 3>에 나타나 있다.

<표 2>의 프로파일링 결과에서 보이듯이 패킷 송신 시에 많은 캐시 미스가 리눅스 커널의 dev_queue_xmit()와 본딩의 bond_xmit_roundrobin()에서 발견되고 있다. 코드 상에서 패킷 송신 시에 bond_xmit_roundrobin()이 호출되고, 라운드로빈 알고리즘을 통하여 패킷을 송신할 네트워크 인터페이스가 선택이 되면 dev_queue_xmit()을 호출하는데, 이때 패

〈표 2〉 기존 본딩 드라이버 프로파일링 결과

module	함수이름	캐시미스	CPI
kernel	dev_queue_xmit(*)	26,435	11.36
kernel	__kfree_skb	18,254	13.60
kernel	sock_wfree	13,582	44.40
kernel	_spin_lock_bh(*)	11,330	5.09
kernel	tcp_v4_rcv	11,078	3.79
kernel	schedule	11,037	3.88
kernel	task_rq_lock	10,583	2.00
kernel	find_busiest_group	10,135	1.53
bonding	bond_xmit_roundrobin(*)	11,619	16.10
bonding	bond_dev_queue_xmit	0	2.50

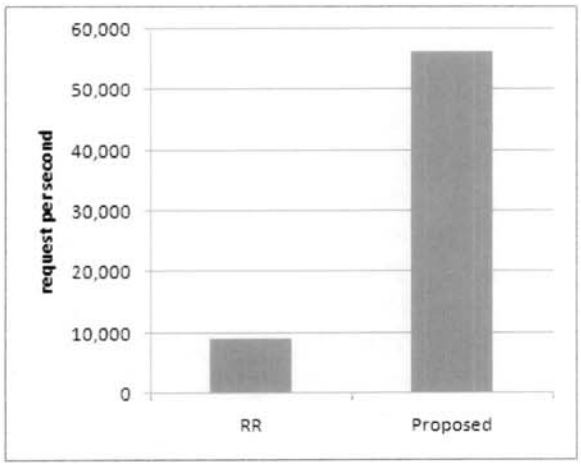
((*) 표시는 본딩 드라이버를 통한 패킷 송신시 캐시미스가 많이 발생하는 곳)

〈표 3〉 개선된 본딩 드라이버 프로파일링 결과

module	Function Name	캐시미스	CPI
kernel	__kfree_skb	27,796	14.44
kernel	tcp_v4_rcv	22,661	4.68
kernel	tcp_current_mss	14,149	5.26
kernel	find_get_page	12,403	30.75
kernel	dev_queue_xmit(*)	11,652	5.28
kernel	ip_queue_xmit	10,743	6.02
kernel	tcp_clean_rtx_queue	9,852	3.24
kernel	put_page	8,527	13.94
bonding	bond_xmit_cpu_bound(*)	582	1.64
bonding	bond_dev_queue_xmit	2	1.32

((*) 제안된 방법을 통해 개선된 곳)

킷 송신을 시도하는 코어와 실제 송신을 담당하는 네트워크 인터페이스와 연결된 코어가 다르기 때문에 많은 캐시 미스와 높은 CPI값을 보이고 있다. 또한 이러한 상황에서 락(lock)으로 자료구조가 보호되기 때문에 _spin_lock_bh() 높은 캐시미스를 보이고 있다.



(그림 4) 본딩 드라이버에 대한 성능 비교

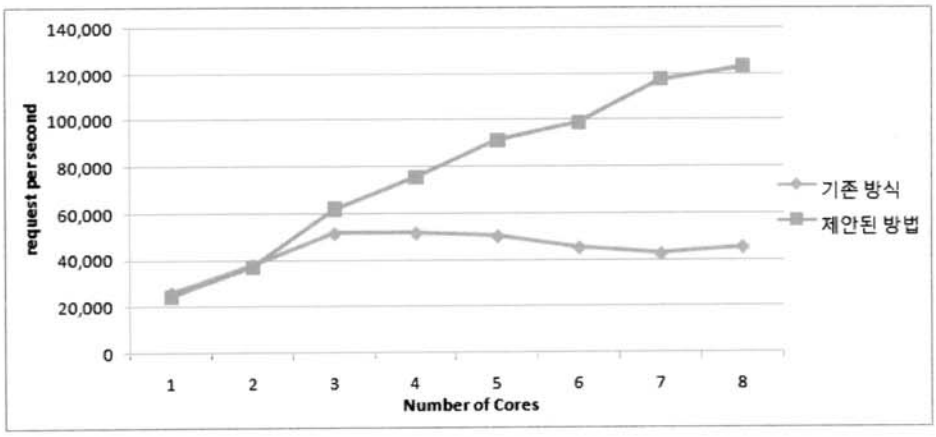
하지만 개선된 본딩 드라이버의 <표 3>에서와 같이 bond_xmit_roundrobin()대신 개선된 방식의 bond_xmit_cpu_bound()가 결과에 나타나고 있으며, 캐시미스가 현저하게 줄어든 것을 확인할 수 있다. 또한 dev_queue_xmit()과 _spin_lock_bh()의 캐시미스도 현저히 감소하여 성능이 개선되었다는 것을 파악할 수 있다.

실제 성능 측정결과에서는 제안된 본딩 드라이버가 동일 조건에서 기존의 본딩 드라이버에 비해 약 6배의 성능향상이 있었다(그림 4). 이는 예상했던 대로 패킷 송신 시에 락(lock)과 공유되는 자료구조로 인해 심각한 성능저하가 일어난다는 사실을 입증해준다.

4.3 제안된 전체 시스템에 대한 실험 및 성능 분석

아파치벤치를 이용하여 기존 방법과 제안된 방법에 대한 접속 속도를 측정하였다. 접속 속도 측정 시에는 웹 서버에 사이즈가 128인 파일을 요청하는 방식으로 코어의 수를 증가시키면서 초당 성공한 접속수를 측정하였다.

(그림 5)에서 보이듯이 기존 방법의 실험 결과는 코어의



(그림 5) 기존방식과 제안된 방식의 접속속도 비교

수가 3개까지는 성능향상을 보이다가 4개째부터는 오히려 성능이 약간 감소하면서 더 이상 증가하지 않았고, 중간 중간에는 성능이 오히려 떨어지는 부분도 발견되었다. 반면에 제안된 방법은 코어의 수가 2개 까지는 기존의 방법과 큰 차이가 없었지만 코어가 3개째부터는 기존 방식에 비해 월등한 성능을 보이고 있으며, 코어를 8개까지 증가시켰을 때 약 300% 정도의 성능 향상을 보이고 있다. 결과적으로 코어의 수가 8개까지 증가하더라도 일정한 성능향상을 보여 제안한 방법에 대한 효용성을 입증하고 있다.

코어의 수가 8개일 때를 기준으로 앞서 설명한 실험을 진행하면서 vtune을 이용하여 프로파일링을 진행하였다. 우선 TCP의 패킷 송수신과 웹 서버가 작동하면서 TCP와 관련된 시스템 콜을 호출하면서 발생하는 캐시미스와 CPI값에 대해 살펴보면 다음과 같다. <표 4>의 프로파일링 결과는 기존 방식과 제안된 방식의 프로파일링 결과 중 캐시미스가 많이 발생한 순대로 50위까지의 순위에서 TCP와 관련된 함수들을 뽑아낸 것이다. 먼저 이들에 대한 전체적인 캐시 미스와 CPI값을 보았을 때, 개선된 방식의 캐시미스 회수가 60,754에서 42,448으로 내려가 30%정도 감소하였으며, CPI값은 5.63에서 2.84로 내려가 상당한 개선효과를 보이고 있다.

또한 캐시미스가 개선된 지역이 초기에 예상했던 대로, TCP의 영역 중 유일한 자료구조를 사용하는 리스닝 소켓을 통해 새로운 소켓을 생성해 내는 과정(①, ②)에서 많이 발생한 것으로 파악이 되었다. 또한 웹 서버관점에서 시스템 콜을 통하여 외부 요청을 수락하는 accept()함수 호출(③)에 또한 단일 리스닝 소켓을 통해서 새로운 소켓을 얻어내는데, 이 지점에서도 많은 성능 저하가 발생하는 것으로 파악되었으며, 제안된 방법을 통해 많은 개선이 되었음을 확인할 수 있다. 하지만 데이터 전송(④)과 접속 종료(⑤)에서는 캐시미스는 제안된 방법이 좀 더 큰 값을 나타내어 개선의 여지가 있음을 보이고 있지만, CPI값이 줄어들어 실제 코드 수행속도는 개선되었을 것으로 판단된다.

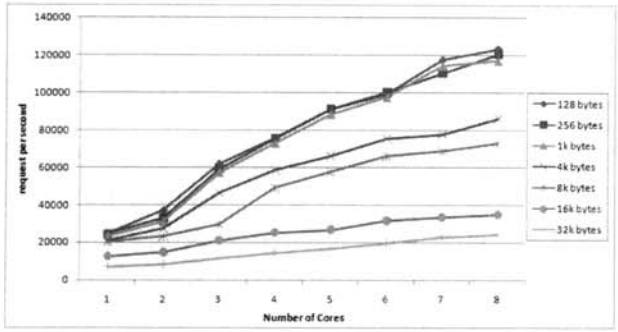
패킷을 네트워크 인터페이스로 내보내는 과정은 본딩 드라이버의 실험결과와 거의 동일한 양상을 보였다. 기존 방식에서는 패킷 송신에 사용되는 dev_queue_xmit()과 _spin_lock_bh()의 캐시미스와 CPI값이 상당히 높게 나타났으며, 개선된 방식에서는 dev_queue_xmit()의 캐시미스가 기존에 비해 약 30%정도, CPI는 10.19에서 6.92로 줄어들었다. 또한, 패킷 송신 시 사용되는 락(lock)인 _spin_lock_bh()의 캐시미스는 약 75% 정도로 감소하였다. 이는 다수의 코어가 네트워크 인터페이스를 접근하기 때문인 것으로 파악할 수 있으나, _spin_lock_bh()의 경우 커널의 다른 영역에서도 사용된다는 점은 감안하여야 한다.

제안된 방법이 접속 속도뿐만이 아니라, 데이터의 전송률

〈표 4〉 기존 방식과 제안된 방식의 프로파일링을 통한 TCP 스택의 성능 비교

테스트 방법		기존 방식		제안된 방식	
용도	함수이름	캐시미스	CPI	캐시미스	CPI
① TCP패킷수신	tcp_v4_rcv	16,499	4.88	9,450	1.95
	tcp_v4_do_rcv	3,567	4.60	114	1.75
② 리스닝 소켓 처리 및 새로운 소켓 생성	tcp_v4_hnd_req	7,157	4.10	203	1.52
	tcp_v4_syn_recv_sock	1,823	3.81	2,242	1.57
	tcp_rcv_state_process	1,920	1.44	2,310	2.60
	__inet_lookup_listener	5,140	15.35	6,475	5.60
③ 접속 수락 (시스템 콜)	tcp_poll	6,758	13.73	102	0.00
	inet_csk_accept	6,316	7.58	6,408	7.39
	inet_accept	2,734	4.35	1,826	4.43
④ 전송 (시스템 콜)	tcp_sendmsg	1,113	3.71	2,209	3.20
	tcp_current_mss	4,846	2.41	5,852	2.15
⑤ 접속 종료 (시스템 콜)	tcp_close	1,995	1.65	2,947	2.14
	tcp_v4_destroy_sock	886	5.64	2,310	2.60
합계		60,754	-	42,448	-
평균		-	5.63	-	2.84

관점에서의 확장성을 측정하였다. 앞서 행한 실험과 유사한 방법으로 진행되었으며, 아파치벤치로 웹서버에 256바이트부터 32k bytes까지 범위의 요청을 만들어 실험을 하였다. 아래 (그림 6)은 그 결과를 나타내는 그래프로 데이터의 사이즈와 무관하게 성능이 증가한다는 것을 확인할 수 있었다.



(그림 6) 제안된 방법에 대한 데이터 크기별 접속 테스트

이상과 같은 측정과 검증작업을 통하여 기존 시스템의 문제점에 대한 사항을 점검하였으며, 리눅스 커널 내부의 네트워크 인터페이스, 소켓 등이 다수의 코어에 의하여 공유가 되고, 이들이 락(lock)으로 보호되는 경우 심각한 성능저하현상을 보인다는 것을 확인했고, 이들을 피할 수 있도록 웹 서버를 운영할 경우 상당한 성능향상과 더불어 코어의 수에 비례하는 선형적인 성능 증가를 얻을 수 있다는 결론을 도출하였다.

5. 결 론

본 논문에서는 네트워크 애플리케이션을 운용하는 리눅스 대칭형 다중 처리 시스템에서 성능상의 문제점을 파악하고 이를 해결 할 수 있는 방법을 제안하였다. 제안된 시스템은 외부로부터의 접속이 패킷 플로우 단위로 다수의 코어에 분산해서 처리할 경우, 네트워크 입출력 성능이 향상될 수 있다는 가정에 출발하여, 기존에 나와 있는 L2 스위치, 본딩 드라이버 등과 애플리케이션의 수정을 통하여 시스템 내부의 네트워크와 관련된 리소스를 각각의 코어단위로 가질 수 있도록 시스템을 설계하였다. 실험을 통한 검증에서 접속속도에서는 약 300%정도의 성능 개선효과가 있었으며, 8개의 코어까지 선형적인 성능 증가효과를 보여 제안한 방법이 실제로 효용성이 있다는 것을 검증하였다.

성능 측정 실험결과를 프로파일링을 통한 분석으로 실제로 네트워크와 관련된 리눅스 커널의 자료구조로 인하여 성능저하를 보이는 코드상의 위치를 확인하였다. 향후에 이러한 자료를 바탕으로 리눅스 커널의 TCP 스택의 리스닝 소켓과 관련한 자료구조들을 코어단위로 생성할 수 있도록 개선된다면 기존보다 나은 성능을 보일 수 있을 것으로 보인다. 또한 네트워크 인터페이스에 있어서도 현실적인 문제를 감안하여 L2 스위치와 개선된 본딩 드라이버를 이용하여 각각의 코어에 패킷을 플로우 단위로 분할하여 송수신을 하도록 하였다. 하지만 이상과 같은 방법으로 소수의 커넥션이 다량의 트래픽을 발생시키는 경우 특정 코어에 부하가 집중되는 현상이 발생할 수 있는데, 이에 대해서는 별도의 연구가 필요하며 코어에 부하분산을 효과적으로 수행할 수 있는 기능이 향후 네트워크 인터페이스에서 지원될 수 있으리라고 판단한다.

하지만 본 논문에서는 TCP에 대한 네트워크 입출력과 관련한 실험만을 위주로 진행하였기 때문에, 실제 웹 서버와 연계되는 별도의 애플리케이션이나 데이터베이스 그리고 파일 시스템과 연계해서 작동하였을 때 발생하는 성능 저하의 문제에 대해서는 고려하지 않았다. 실제 시스템이 복합적으로 작동할 경우에는 실험 결과와 같은 성능 향상은 얻을 수 없을 것으로 보이며, 각각의 애플리케이션의 특성에 맞춘 성능 개선 방법 또한 필요하다고 볼 수 있다. 하지만 다수의 커넥션을 처리를 위한 네트워크 입출력이 관건인 애플리케이션에서는 이 실험 결과만을 토대로도 높은 성능향상을 얻을 수 있을 것으로 기대한다.

마지막으로 본 연구결과의 목표는 일반적인 사용자들이 멀티코어 시스템에서 기존의 수많은 애플리케이션을 기존에 사용하던 방식 그대로 운영하였을 때, 코어의 수에 비례하는 성능 개선효과를 얻을 수 있도록 하는 것이다. 하지만 제안된 방법에서는 아직까지 일반적인 사용자들이 적용하는 것은 무리일 것으로 판단한다. 향후 연구로는 본 논문에서 나온 실험결과를 토대로 현재 커널 내부의 TCP/IP 스택에 일부 변화를 가하여 애플리케이션의 운영이나 사용방법 그리고 코드를 변경하지 않은 상태에서 성능 개선을 얻는 것을 생각해볼 수 있다.

참 고 문 헌

[1] T. Tian and C-P Shih, "Software Techniques for Shared-Cache Multi-Core Systems", Intel Software Network, http://softwarecommunity.intel.com/articles/eng/2760.htm

[2] Intel® white paper, Supra-linear Packet Processing Performance with Intel® Multi-core Processors," www.intel.com/technology/advanced_comm/311566.htm, 2006.

[3] Intel® Excecutive Summary, "Accelerating Security Applications with Intel® Multi-core Processors", www.intel.com/technology/advanced_comm/314312.htm, 2006.

[4] E. Lemoine, C. Pham and L. Lef'evre, "Packet Classification in the NIC forImproved SMP-based Internet Servers", IEEE Proceedings of the InternationalConference on Networking (ICN 2004),Guadeloupe, French Caribbean, Feb., 2004.

[5] Yi, Z. and Waskiewicz, P.J., 2007. Enabling Linux Network Support of Hardware Multiqueue Devices. Proc. of 2007 Linux Symposium, Ottawa, Canada, June, 2007, 305-310.

[6] A. Muir and J.Smith. "AsyMOS: An asymmetric multiprocessor OS", In Proc. of OPENARCH'98, 25-34, April, 1988.

[7] G. Regnier, D.Minturn, G. McApline, V. Saletore, A.Foong. "ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine", IEEE Micro, Jan., 2004.
[8] Bryan Veal, Annie Foong, "Performance Scalability of a Multi-core Web Server", ANCS 2007.
[9] IRQ affinity, http://www.kernel.org/doc/Documentation/IRQ-affinity.txt, Linux Kernel Documentation
[10] Annie Foong, Jason Fung, Donald Newell, Seth Abraham, Peggy Irelan, Alex Lopez-Estrada: Architectural Characterization of Processor Affinity in Network Processing. ISPASS 2005: 207-218.
[11] Christian Benvenuti, Chapter 10 Frame Receptions, Understanding Linux Network Internals, O'Reilly &Associates, 2005.
[12] Paul E. McKenney. RCU vs. locking performance on different CPUs. Inlinux.conf.au, Adelaide, Australia, January, 2004.
[13] M. Bjorkman and P.Gunningberg. Locking effects in multiprocessor implementation of protocols. In Proc. ACM SIGCOMM '93 Conference, pages 74-83, October, 1993.
[14] bonding driver, http://www.kernel.org/doc/Documentation/networking/bonding.txt
[15] Link Aggregation, http://standards.ieee.org/reading/ieee/std_public/new_desc/lanman/ restricted/802.3ad-2000.html
[16] Process Affinity, http://www.kernel.org/doc/man-pages/online/pages/man2/sched_ getaffinity.2.html
[17] Lighttpd, http://www.lighttpd.net/
[18] apache bench, http://httpd.apache.org/docs/2.0/programs/ab.html
[19] vtune, http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm

권 희 웅

e-mail : didorito@q.ssu.ac.kr
1997년 숭실대학교 정보통신전자공학부(학사)
1999년 숭실대학교 전자공학과(석사)
1999년~현 재 숭실대학교 정보통신전자공학부 박사과정
관심분야 : 네트워크 및 어플리케이션에 대한 부하 분산, 가속, 보안

정 형 진

e-mail : realbright@q.ssu.ac.kr
2007년 숭실대학교 정보통신전자공학부(학사)
2007년~현 재 숭실대학교 정보통신전자공학부 석사과정
관심분야 : 네트워크 컴퓨팅 및 보안

곽 후 근

e-mail : gobarian@q.ssu.ac.kr
1996년 호서대학교 전자공학과(학사)
1998년 숭실대학교 전자공학과(석사)
1998년~2006년 숭실대학교 전자공학과(박사)
1998년 8월~2000년 7월 (주) 3R 부설 연구소 주임연구원
2006년 3월~현 재 숭실대학교 정보통신전자공학부 postdoc
관심분야 : 네트워크 컴퓨팅 및 보안

김 영 종

e-mail : opensys@q.ssu.ac.kr
1996년 7월~1998년 4월 ㈜한글과컴퓨터/연구원
2000년 9월~2004년 11월 ㈜캐스트와이즈/대표이사
2006년 1월~2008년 4월 ㈜하우리/대표이사
2007년 7월~2009년 3월 열린사이버대학교/정보지원처장
2007년 12월~2009년 3월 오픈소스커뮤니티연구소/소장
2007년 3월~현 재 숭실대학교 정보통신전자공학부 석사과정
관심분야 : 네트워크 컴퓨팅 및 보안

정 규 식

e-mail : kchung@q.ssu.ac.kr

1979년 서울대학교 전자공학과(공학사)

1981년 한국과학기술원 전산학과(이학석사)

1986년 미국 University of Southern California(컴퓨터공학석사)

1990년 미국 University of Southern California(컴퓨터공학박사)

1998년 2월~1999년 2월 미국 IBM Almaden 연구소 방문연구원

1990년 9월~현 재 숭실대학교 정보통신전자공학부 교수

관심분야 : 네트워크 컴퓨팅 및 보안