KIPS Tr. Comp. and Comm. Sys.
Vol.3, No.8 pp.261~270 pISSN: 2287-5891

http://dx.doi.org/10.3745/KTCCS.2014.3.8.261

A Performance Improvement of Linux TCP Networking by Data Structure Reuse

Seokkoo Kim† · Kyusik Chung††

ABSTRACT

As Internet traffic increases recently, much effort has been put on improving the performance of a web server. In addition to hardware side solutions such as replacement by high-end hardware or expansion of the number of servers, there are software side solutions to improve performance. Recent studies on these software side solutions have been actively performed.

In this paper, we identify performance degradation problems occurring in a conventional TCP networking reception process and propose a way to solve them. We improve performance by combining three kinds of existing methods for Linux Networking Performance Improvement and two kinds of newly proposed methods in this paper. The three existing methods include 1) an allocation method of a packet flow to a core in a multi-core environment, 2) ITR(Interrupt Throttle Rate) method to control excessive interrupt requests, and 3) sk_buff data structure recycling. The two newly proposed methods are fd data structure recycling and epoll_event data structure recycling.

Through experiments in a web server environment, we verify the effect of our two proposed methods and its combination with the three existing methods for performance improvement, respectively. We use three kinds of web servers: a simple web server, Lighttpd generally used in Linux, and Apache. In a simple web server environment, fd data structure recycling and epoll_event data structure recycling bring out performance improvement by about 7 % and 6%, respectively. If they are combined with the three existing methods, performance is improved by up to 40% in total. In a Lighttpd and an Apache web server environment, the combination of five methods brings out performance improvement by up to 36% and 20% in total, respectively.

Keywords : Linux TCP Networking, Performance Improvement, Multi-Core, Data Structure Recycling

자료 구조 재사용을 이용한 리눅스 TCP 네트워킹 성능 개선

김 석 구† · 정 규 식††

요 약

최근 인터넷 트래픽이 증가하면서 웹 서버의 성능 향상에 많은 노력들을 기울여왔다. 고사양 하드웨어로의 교체 또는 서버 수의 증설과 같은 하드웨어 측면 해결방법 외에 소프트웨어 측면의 해결 방법들이 있는데 최근 이에 대한 연구들이 활발히 진행되고 있다.

본 논문에서는 기존 TCP 네트워킹 수신과정에서 발생하는 성능 저하 문제점들을 파악하고 이를 해결할 수 있는 방법을 제안한다. 리눅스 TCP 네트워킹 성능 개선에 관한 기존 방법 세 가지와 본 논문에서 새로 제안하는 두 가지 방법을 통합 적용하여 성능을 향상시킨다. 기존 개선 방법들로는 멀티코어 환경에서 패킷을 흐름단위로 코어에 할당하는 방법, 과도한 인터럽트 요청을 조절하는 ITR(Interrupt Throttle Rate) 방법, sk_buff 자료구조 recycling 방법이다. 본 논문에서 새로 제안하는 방법은 fd 자료구조 recycling 방법과 epoll_event 자료구조 recycling 방법이다.

웹 서버 환경에서 실험을 통해 본 논문의 제안방법들의 성능 개선효과, 또한 기존방법들과의 통합 적용했을 경우 성능 개선효과를 검증한다. 웹 서버로는 간단한 웹 서버, 리눅스에서 일반적으로 사용하는 Lighttpd와 Apache 웹 서버를 사용한다. 간단한 웹 서버 환경에서 본 논문에서 제안한 fd 재사용과 epoll_event 재사용을 적용할 경우 성능이 각각 7%와 6% 개선되고, 이 두 가지 방법을 기존의 세 가지 방법과 통합하여 적용한 경우 성능이 총 40%까지 개선된다. Lighttpd와 Apache 웹 서버 환경에서 다섯 가지 통합 방법을 적용한 경우 성능이 각각 총 36%, 20%까지 개선된다.

키워드 : 리눅스 TCP 네트워킹, 성능 향상, 멀티코어, 자료구조 재사용

※ 이 논문은 펌킨네트웍스(주) 지원을 받아 연구되었음.
† 비 회 원 : 숭실대학교 정보통신공학과 석사
†† 정 회 원 : 숭실대학교 정보통신전자공학부 교수

Manuscript Received : January 2, 2014
First Revision : June 9, 2014; Second Revision : July 7, 2014
Accepted : July 9, 2014
* Corresponding Author : Kyusik Chung(kchung@q.ssu.ac.kr)

1. 서 론

인터넷 속도의 발달과 스마트 기기의 보급 확대로 네트워크 트래픽량이 급격하게 상승하고 있다. 이러한 환경에서 인터넷 서비스를 담당하는 웹 서버의 성능 향상의 필요성은 계속 높아지고 있다. 웹 서버의 성능을 높일 때 신규 고성능 서버로의 하드웨어 교체 방법이 있으나 제한적이다. 하드웨어 측면뿐 아니라 소프트웨어 측면에서도 성능 개선이 이루어져야 최적의 웹 서버 성능 향상이 가능하다.

최근에는 소프트웨어적인 측면의 웹 서버 성능 개선을 위한 연구들이 많이 진행되고 있다. 대용량 트래픽 상황에서 웹 서버의 성능 저하 문제점은 주로 서버와 클라이언트 사이의 TCP 네트워킹 수신과정에서 많이 지적되고 있다. 멀티코어 환경에서, TCP/IP 스택 과정을 수행하는 코어와 애플리케이션을 수행하는 코어가 서로 다를 경우 캐시미스와 같은 이유로 발생하는 성능 저하 문제[1, 2], 패킷이 서버에 수신될 때마다 인터럽트를 이용하여 처리하는 시스템에서 대용량 패킷을 처리할 경우 많은 인터럽트 횟수로 인해 발생하는 성능 저하 문제[4], 사용자가 서버에 접속을 하였다가 접속을 해제할 때마다 자료구조체를 위한 메모리 할당 및 해제를 반복하게 되는데, 잦은 메모리 할당과 해제로 인한 성능 저하 문제[9]들이 지적되고 이에 대한 각각의 해결책들을 제시하고 있다.

본 논문에서는 자료구조체 사용에 관한 새로운 방법을 제안하며, 기존 연구 방법들을 통합 적용하여 웹 서버에서의 가능한 큰 성능 향상을 얻는 것을 목표로 하고 있다. 본 논문의 구성은 다음과 같다. 2장에서는 기존 연구를 소개하고, 3장에서는 제안된 리눅스 TCP 네트워킹 성능 개선 방법에 대해 소개한다. 4장에서는 실험 및 결과를 소개하고 마지막으로 5장에서는 결론으로 끝을 맺는다.

2. 연구 배경

2.1 기본적인 TCP 동작 흐름 과정

서버와 클라이언트 사이의 기본적인 TCP 동작 흐름을 그림 1과 같이 나타낼 수 있다. 크게 세 과정으로 나눠 볼 수 있으며 연결 과정((1),(2),(3)), 요청 응답 과정((4),(5): 이 과정은 여러 번 반복가능), 연결 종료 과정((6),(7),(8))이다. 세부 과정은 다음과 같다.

(1) 클라이언트가 서버에게 연결 요청(SYN)을 한다.
(2) 서버가 연결 요청을 수락(SNY-ACK)한다.
(3) 클라이언트가 연결 요청에 대한 수락 메시지를 받았다는 확인(ACK)을 보내고,
(4) 요청(Request)을 한다.
(5) 서버가 클라이언트의 요청에 대한 응답(Response)를 보내고,
(6) 연결 종료(FIN) 메시지를 보낸다.
(7) 연결 종료 메시지를 받은 클라이언트도 연결 종료 메시지를 보낸다.
(8) 서버가 연결 종료 메시지에 대한 응답을 보내고 완전히 연결이 종료된다.

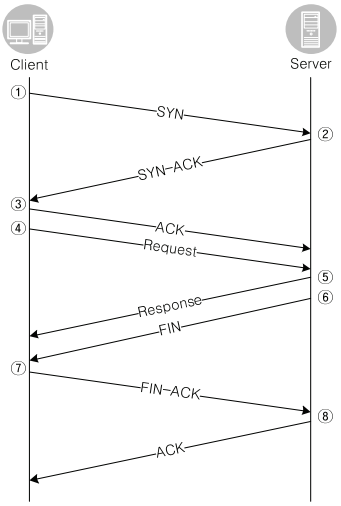

Fig. 1. Basic TCP operation flow

2.2 기존 TCP 네트워킹 패킷 수신과정[1, 2]

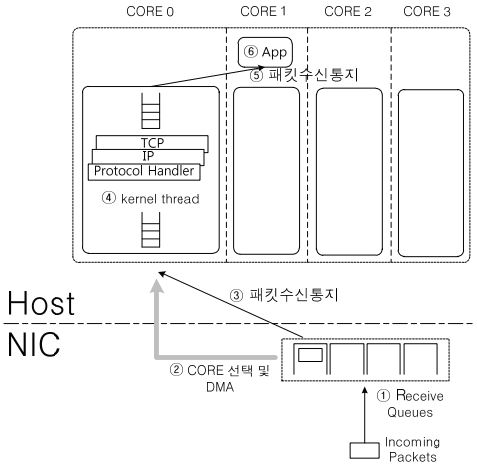

Fig. 2. Packet receiving process

일반적인 멀티코어 서버 환경에서 패킷 수신 과정은 그림 2와 같이 간략하게 표현할 수 있다. 그림 2를 크게 두 부분으로 나누어 설명할 수 있는데, 하나는 NIC 부분으로 네트워크카드에서 패킷을 수신하는 디바이스 드라이버의 동작과정이고, 다른 하나는 HOST 부분으로 디바이스 드라이버를 제외한 커널의 동작과정이다. 패킷 수신 과정은 다음과 같다.

(1) 수신된 패킷을 가장 먼저 수신 큐에 저장한다.
(2) NIC 드라이버 설정에 따라 멀티코어 중 하나의 코어를 선택하고, DMA를 이용하여 커널 영역에 복사한다.
(3) DMA를 이용한 복사 과정이 끝나면, 앞선 과정에서 선택된 코어에게 패킷 수신 통지를 한다.

(4) 패킷 수신 통지를 받은 코어는 수신한 패킷을 처리하기 위해 커널 스레드(kernel thread)를 구동시켜 패킷 단위로 프로토콜 핸들러(protocol handler)를 호출하며, 프로토콜 핸들러 내부에서 TCP/IP에 대한 스택에 대한 핸들러를 호출하여 적절한 처리를 하게 하며, 로컬인 패킷의 경우 사용자영역으로 패킷을 복사한다.
(5) 사용자영역으로 패킷 복사 과정이 끝나면, 해당 애플리케이션에게 패킷 수신 통지를 한다.
(6) 패킷 수신 통지를 받은 애플리케이션은 최종적으로 패킷을 처리하게 된다.

2.3 패킷 수신 과정에서 자주 사용되는 자료 구조체

패킷을 수신하는 과정 중에 자주 사용되는 자료 구조체들로는 sk_buff, fd, event_poll 등이 있다.

첫 번째, sk_buff 소켓 버퍼는 네트워크로 수신되는 패킷을 나타내는 자료 구조로서, 네트워크 서브 시스템 전반에서 사용되는 중요한 구조체이다. 이 구조체는 패킷이 수신될 때마다 할당, 해제된다. 그림 2의 패킷 수신 과정에서 살펴보면 과정(1)에서 패킷이 수신될 때마다 할당된다. 이후 해당 패킷에 대한 처리가 완료되면 해제가 된다. sk_buff 자료구조체 구조는 그림 3과 같다.

두 번째, 파일 디스크립터인 fd 자료 구조체는 애플리케이션 단에서 사용되는데 커널 내부에서는 그림 4와 같은 자료 구조를 가진다[3]. fd는 애플리케이션 단에서 여러 클라이언트가 존재할 때 클라이언트들을 구별하는 값으로 볼 수 있다. 처음 클라이언트가 서버에게 연결을 요청할 때 하나

```
Struct sk_buff{
    next,prev
    struct sock *sk;
    …
    dev
    /* TP layer header */
    union { th, uh, icmph, …}h;
    /* Network layer header */
    union { iph, ipv6h, arph, …}nh
    /* Data Link layer header */
    union { raw }mac;
    struct dst_entry *dst
    …
    data, head, tail, len
    …
}
```

Fig. 3. sk_buff data structure

의 fd값을 할당받고, 이후부터 수신되는 메시지는 fd를 이용하게 된다. 다시 말하면, 그림 1에서의 과정(2) 요청을 수락할 때 해당 클라이언트를 구별하기 위해 fd를 하나 할당하게 된다. 이렇게 하나씩 할당한 fd를 이용하여 서버는 클라이언트들을 구별하면서 동시 통신을 수행하게 된다. 할당한 fd는 그림 1에서의 과정(8)에서 연결을 종료하면서 해제하게 된다.

세 번째, epoll_event는 epoll 동작 과정 중에 사용되는 구조체이다. 그림 2의 과정(5)를 살펴보면 커널이 애플리케이션에게 패킷 수신을 통보하는데, I/O 멀티플렉싱 기법으로 구현되어 있는 경우 애플리케이션은 어떤 클라이언트 패킷이 도착했는지 직접 찾아야 한다. 이때 사용하는 방법으로 select, poll, epoll이 존재한다. Select가 가장 먼저 개발되었

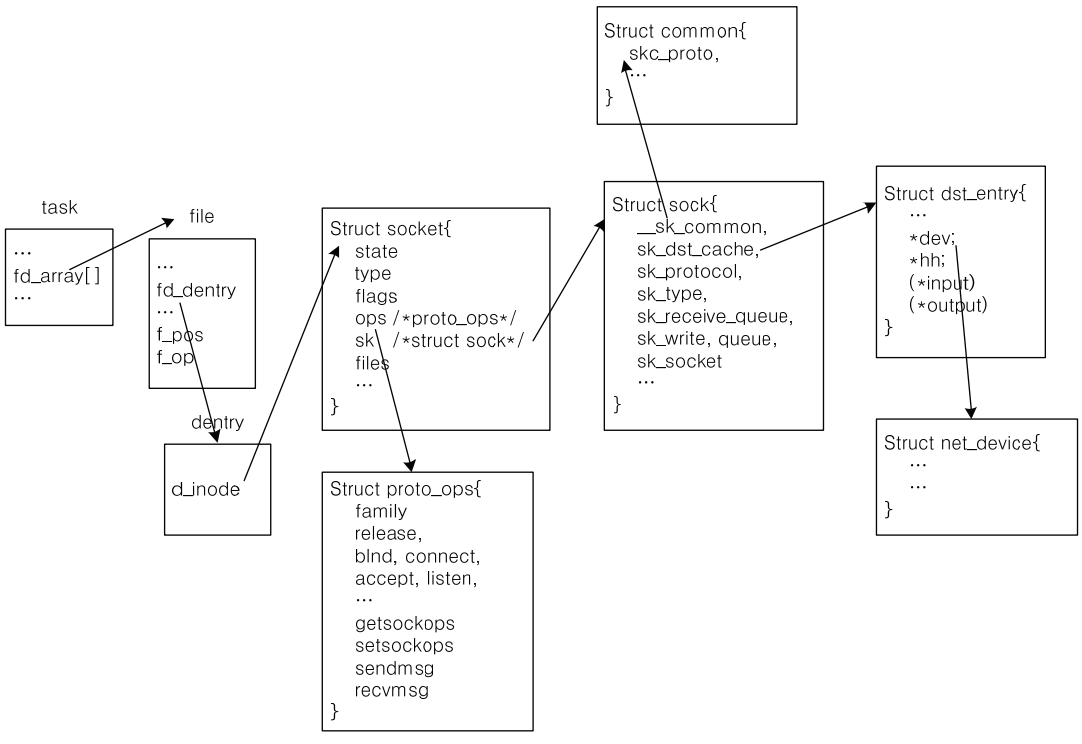

Fig. 4. Socket and sock data structure

고 이후에 개선된 방법이 poll이며 가장 최근에 개발된 게 epoll 방법이다. select과 poll 방법의 경우 애플리케이션이 관찰 대상을 관리하고 매번 커널에게 관리 대상들을 알려주지만 epoll 방법의 경우 애플리케이션이 관찰대상을 최초 한 번만 커널에게 넘겨주면 커널에서 관리한다. 따라서 epoll 방법의 경우, 커널과 사용자 간의 통신 부하를 대폭 줄일 수 있다. 이때 관찰 대상을 등록 해제할 때 사용하는 구조체가 epoll_event이다.

앞서 설명한 세 구조체는 패킷 수신과정뿐만 아니라 패킷 송신과정에서도 사용된다.

2.4 기존 TCP 네트워킹 성능 개선 방법

기존의 패킷 수신 과정에 있어서 성능 저하 문제점들을 지적할 수 있다.

첫 번째, 멀티코어 환경에서 패킷 흐름에 따른 성능 저하 문제이다. 하나의 패킷 수신 과정에서 모든 과정이 하나의 동일한 코어에서 수행될 때 성능 저하가 최소화될 수 있다. 그림 2에서 TCP/IP 스택 과정을 수행하는 코어와 애플리케이션이 수행 중인 코어가 다를 경우, 캐시미스 증가와 락(Lock)사용으로 성능 저하가 발생한다[1,2].

두 번째, 대용량 패킷을 처리하는 시스템에서 생기는 성능 저하 문제이다. 그림 2의 과정(3)에서 기본적으로 패킷이 수신될 때마다 인터럽트가 발생한다. 패킷 수신이 많아질수록 인터럽트 핸들러를 수행하고 스케줄링하는데 소요하는 시간이 증가하게 된다. 계속적으로 패킷 수신이 많을 경우, 패킷 드롭이 발생할 수 있으며 일시적으로 다운될 수 있다[4].

세 번째, 자료 구조체를 위한 메모리를 매번 새로 할당하고 해제하는 과정에 발생하는 성능 저하 문제이다. 이 과정은 2.3에 설명한 세 구조체에 대한 부분이다. Sk_buff의 경우 그림 2의 과정(1)에서 매번 새로운 패킷이 수신될 때마다 새로 할당, 해제하는 과정을 거치고, fd의 경우 그림 1의 과정(2)에서 사용자가 접속할 때마다 새로 할당되고, 그림 1의 과정(8)에서 접속을 끊을 때 해제된다. epoll_event의 경우 fd와 마찬가지로 그림 1의 과정(2)에서 사용자가 접속할 때마다 새로 할당하여 등록하고 그림 1의 과정(8)에서 사용자가 접속을 끊을 때 해제된다. 이와 같은 잦은 메모리 할당과 해제는 성능의 저하를 발생시킨다.

이러한 문제점들에 대한 기존 연구들을 살펴보면 다음과 같다.

첫 번째, 멀티코어 환경에서 성능 향상 시도는 전용하드웨어부터 소프트웨어 기법까지 다양한 방면으로 시도되었다. Intel의 snort에 대한 연구[5, 6]는 하나의 코어가 패킷을 수신하고 수신된 패킷들을 TCP 플로우 단위로 각각의 코어에 분산하는 구조를 채택하여 성능을 개선하였으며 SUN의 NIC을 통한 TCP 플로우 단위의 부하분산[7]을 통해 성능을 개선하였다. [2]에서는 코어별 전용 NIC를 두고 NIC들을 하나의 L2스위치로 연결하여 흐름(Flow) 단위의 부하분산을 시도하여 성능을 개선하였다. 현재에는 몇몇 이더넷카드와

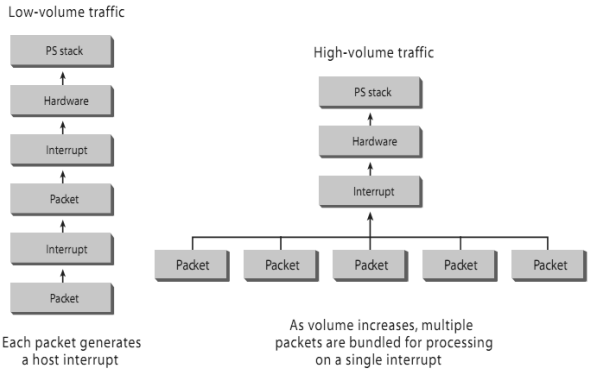

Fig. 5. Host Interrupt Modulation

해당 드라이버에서 옵션을 통해 흐름 단위로 패킷을 조절할 수 있다[8].

두 번째, 대용량 패킷을 처리하는 시스팀에서 생기는 성능 저하 문제는 Intel에서 배치(Batch) 처리를 적용하여 최소화하였다. 그림 5에 보인 바와 같이, 낮은 트래픽 상황에서는 각각의 패킷 수신마다 인터럽트를 발생시키고, 높은 트래픽 상황에서는 여러 패킷을 묶어 하나의 인터럽트 과정을 통해 처리하여, 인터럽트 핸들러를 수행하고 스케쥴링하는데 걸리는 부하를 최소화하였다[4].

세 번째, 잦은 메모리 할당으로 인한 성능 저하 문제점 중 sk_buff 구조체의 경우 sk_buff 자료 구조체를 해제하지 않고 보관하였다가 패킷 수신시 재사용하는 방법으로 부하를 최소화하였다[9]. sk_buff 구조체 외에 fd 관련 구조체 및 epoll_event 구조체 재사용을 추가로 제안하고 기존 방법들과 통합 적용을 통해 TCP 네트워크 성능을 개선하였다[16]. 본 논문은 위 논문[16]의 확장 버전으로, 제안 방법들에 대하여 간단한 웹 서버 환경뿐만 아니라 Lighttpd 및 Apache 웹 서버 환경에서도 실험을 수행하여 제안 방법의 유효성을 다각도로 검증하였다.

다음의 표 1은 기존 연구들을 비교한 결과이다.

Table 1. Comparison of Existing Methods

기존연구	성능 개선점	성능 향상도
Performance Improvement of Linux TCP Networking based on Flow-Level Parallelism in a Multi-Core System [2]	멀티코어 시스템에서 흐름 수준 성능 개선 반영	300%
Interrupt Moderation Using intel GbE Controllers[4]	대용량 패킷 처리 시스템에서 성능 개선 반영	데이터 없음
A high-end Linux based Open Router for IP QoS networks : tuning and performance analysis with internal (profiling) and external measurement tools of the packet forwarding capabilities [10]	sk_buff 자료구조 재사용으로 성능 개선 반영	6%

3. 제안된 리눅스 TCP 네트워킹 성능 개선

기존 연구들은 하나의 성능 저하 요소를 부각시키고 개선 방법을 제안하고 있다. 이 방법들 중에 서로 독립적으로 구성되어 동시 적용이 가능한 부분이 있다면 더 큰 성능 향상을 기대할 수 있다.

본 논문에서는 기존 연구방법들 중 서로 영향을 주지 않는 3개의 방법을 통합 적용하며, 추가적으로 2개의 방법을 제안하여 총 5개의 방법을 통합 적용한다. 크게 멀티코어에서 흐름 수준의 성능 개선, 대용량 패킷 처리시스템에서 성능 개선, 그리고 자료구조 재사용을 이용한 성능 개선으로 분류하여 설명한다.

3.1 멀티코어에서 흐름 수준의 성능 개선[2]

```
Intel(R) Ethernet Flow Director
-------------------------------
NOTE: Flow director parameters are only supported on kernel versions 2.6.30
or later.

Supports advanced filters that direct receive packets by their flows to
different queues. Enables tight control on routing a flow in the platform.
Matches flows and CPU cores for flow affinity. Supports multiple parameters
for flexible flow classification and load balancing.

Flow director is enabled only if the kernel is multiple TX queue capable.

An included script (set_irq_affinity) automates setting the IRQ to CPU
affinity.

You can verify that the driver is using Flow Director by looking at the counter
in ethtool: fdir_miss and fdir_match.

Other ethtool Commands:
To enable Flow Director
        ethtool -K ethX ntuple on
To add a filter
        Use -U switch. e.g., ethtool -U ethX flow-type tcp4 src-ip
        192.168.0.100 action 1
To see the list of filters currently present:
        ethtool -u ethX

The following two parameters impact Flow Director.
```

Fig. 6. Intel® Ethernet Flow Director

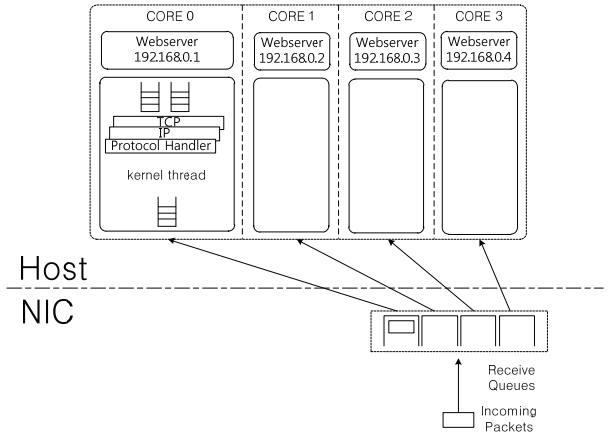

Fig. 7. Flow director perfect filter mode

코어마다 웹 서버를 각각 구동시키고 Flow director를 이용하여 해당 패킷의 수신 전 과정을 하나의 코어에서 처리하게 하여 캐시 미스와 락 사용을 최소하였다. Flow director의 경우 그림 6과 같이 ethtool을 이용하여 설정이 가능하다. 그림 7과 같이 코어마다 송수신 큐를 할당하고 IP 또는 PORT 정보를 이용하여 특정 큐에 수신된 패킷을 할당한다. 각각 큐에 저장된 패킷들은 담당 코어가 처리한다. 예를 들어 192.168.0.1로 수신되는 패킷들은 코어0에 할당된 큐로 저장되며, 코어0에서 패킷 수신 과정을 처리한다. 이후 최종적으로 코어0에서 동작 중인 웹 서버가 패킷을 처리한다. 이처럼 패킷 수신의 전 과정이 하나의 코어에서 처리가 되고, 캐시 미스를 최소화한다. 단, 웹 서버 부하분산기가 따로 존재하여 서버마다 패킷이 균등하게 분배되는 상황을 가정하여 설계하였다.

3.2 대용량 패킷 처리시스템에서 성능 개선[4]

그림 5와 같이 동작하며, 성능 개선을 위해 ITR(Interrupt Throttle Rate)를 조절할 수 있다. 표 2는 ITR 값에 대한 성능 비교표로 보이는 바와 같이 ITR 값이 낮을수록 CPU 사용률이 낮아져 더 많은 트래픽을 처리할 수 있는 반면, Latency는 높아지는 것을 알 수 있다. 본 논문에서는 서버의 트래픽 처리량을 최대로 높이기 위해 ITR를 1,000으로 설정하여 사용한다. ITR 값이 1,000일 때 최고 성능이 나오는지를 검증하기 위해 실험 4.2절에서 보이는 바와 같이 ITR 값에 따른 CPS 값을 측정하였다.

Table 2. Experiment Result of ITR[4]

Value	ITR	Latency	Throughput	CPU Utilization
Extreme	1,000	Highest	High	Lowest
High	1,900	Higher	High	Lower
Medium	4,000	High	High	Low
Low	10,000	Low	High	High
Minimal	19,000	Lower	High	Higher
Off	No Limit	Lowest	Varies	Highest

3.3 자료구조 재사용을 이용한 성능 개선

세 가지 자료구조체를 재사용하였다. 우선 sk_buff 구조체를 재사용하여 메모리 할당에 대한 부하를 줄인 기존 연구방법을 적용하였다[3]. 그리고 oprofile 분석 결과를 토대로 사용자가 접속할 때마다 할당되는 fd와 epoll 관련 구조체 epoll_event 부분을 재사용하는 것을 추가하였다.

3.3.1 Sk_buff recycling[9]

패킷을 수신할 때마다 패킷 정보를 저장하기 위해 새로 할당되는 sk_buff를 사용이 끝난 후에 메모리 해제를 하지 않고 새로운 패킷 수신에 재사용한다. 재사용하는 과정으로는 일정량을 미리 할당해 풀 방식으로 관리하고 필요할 때마다 하나씩 꺼내 쓰고 다 사용한 후에 다시 풀로 반환하는 방식으로 동작한다. 이를 위해서는 디바이스 드라이버의 내부적인 알고리즘만 수정하여 사용이 가능하다.

3.3.2 fd 재사용

그림 4의 구조체들이 매번 새로 할당되면서 발생하는 부하를 최소화하기 위해 3.3.1의 sk_buff recycling과 동일한 구조를 사용하였다. 일정량을 미리 할당해 풀 방식으로 관리하면서 필요할 때 사용하고 반환하는 방식으로 변경하였다. fd 할당의 경우 그림 1의 과정(2)에서 accept4() 함수를 통해 수행된다. fd 재사용을 하기 위해서 프로그램 시작부분에서 일정 크기의 fd 풀을 초기화하고, 프로그램이 종료할 때에는 fd 풀을 해제한다. 그리고 기존 accept4() 함수에는 그림 8과 같이 flags 인자가 존재하여 옵션으로 사용할 수 있다. 해당 옵션에 재사용 옵션을 추가하고 옵션에 대한 처리는 커널에서 구현하였다.

```
#include <sys/socket.h>

int accept4(int sockfd, struct sockaddr *addr,
            socklen_t *addrlen, int flags);
```

Fig. 8. Accept4 function

3.3.3 epoll_event 재사용

한번 등록된 epoll_event를 해제하지 않고 재사용하여, 새로 메모리를 할당하는 부하를 최소화하였다. 그리고 새로 할당된 epoll_event의 경우 관리 대상에 등록, 해제할 때 커널 함수를 호출한다. 하지만 해제과정이 없어지면서 호출 횟수 또한 절반으로 줄었다. 또한 관리 대상에 등록, 해제할 때 락을 사용하여 그림 9와 같이 성능 저하 문제가 발생할 수도 있지만 등록된 이벤트를 재사용함으로써 락의 사용 또한 줄어들었다. 단, epoll_event 구조체의 경우 앞선 3.3.2절의 fd에 종속적으로 fd 재사용이 선행되어야만 epoll_event 구조체를 재사용할 수 있다.

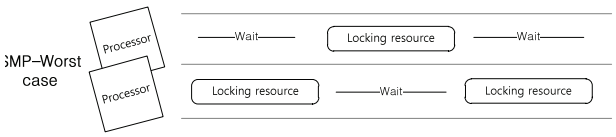

Fig. 9. Memory access through lock

실제 구현 방법으로는 기존에 epoll_event를 등록 또는 해제할 때 사용하는 epoll_ctl에 옵션을 추가하였다. 기존에는 그림 10과 같이 op 인자가 존재하고 EPOLL_ADD 또는 EPOLL_DEL을 입력하여 등록과 해제를 결정한다. Epoll_event를 재사용하기 위하여 EPOLL_POOL이라는 인자를 추가하고, 커널 내부를 수정하였다. 커널 내부에서는 재사용할 수 있는 epoll_evet가 존재하지 않을 경우 기존 EPOLL_ADD 동작과 동일하게 동작하며 재사용할 수 있는 epoll_event가 존재할 경우에 해당 epoll_event를 재사용하게 된다.

```
#include <sys/epoll.h>

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

Fig. 10. epoll_ctl function

4. 실험 및 토론

4.1 실험 환경

실험 환경은 그림 11과 같이 구성하였으며, 서버의 설정을 변경하여 본 논문의 모든 실험을 진행하였다.

Fig. 11. Test Environment

실험에 사용한 서버와 클라이언트 사양은 표 3과 같다.

Table 3. Experimental Environment

Server/Client common	CPU	Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
	RAM	8G bytes
	NIC	Intel x520 10gigabit NIC
	OS	ubuntu 13 (Linux kernel 3.4.11)
	Driver	Ixgbe-3.10.17
Client	Stress tool	AB(Apache Benchmark 2.3)
Server	Web server	Simple webserver, Lighttpd-1.4.31, Apache-2.4.4

서버와 클라이언트 모두 동일한 하드웨어를 이용하여 진행하였다. 코어의 경우 I7-2600(Quad Core)에서 HT(Hyper Thread) 기능을 사용하여 8개 코어로 구성된 시스템을 사용하였으며, Intel x520 10gigabit NIC는 듀얼(Dual) 포트로 구성되어있지만 한 포트만 사용하여 구성하였다. 각각의 서버와 클라이언트는 리눅스 커널 3.4.11 버전을 사용하였다. 테스트 클라이언트로는 아파치 웹 서버에 포함된 성능 측정도구인 아파치벤치(Apache Bench)[11]를 사용하였다. 아파치벤치는 기본적으로 HTTP 프로토콜을 이용하여 초당 접속수인 CPS(Connection Per Second)를 측정하기 위해 사용된다. 측정 결과는 아파치벤치를 8개의 코어에서 동시에 수행시켜 나온 CPS값을 합산하여 사용하였다.

서버애플리케이션으로는 간단한 웹 서버, 기존 리눅스에서 널리 사용 중인 Lighttpd, Apache 웹 서버 세 종류를 사용하였다. 간단한 웹 서버는 본 실험을 위해 본 연구자들이 개발한 서버이다. 기본적으로 I/O 멀티플렉싱 기법을 사용하여 구현하였고 Epoll을 사용하는 단일 스레드 모델이다. 웹 서버의 가장 기본 동작인 클라이언트가 요청한 페이지를 보내주는 동작만 구현하고, 그 외 기능들은 구현하지 않았다. Lighttpd는 1.4.31 버전을 사용하였고 Apache는 2.4.4 버전을 사용하였다. Ligttpd와 Apache 웹 서버의 경우 select, epoll 등을 지원하며 해당 버전에서는 epoll이 기본 설정으로 사용되고 있고, 설정 파일의 경우 기본 설정 파일을 그대로 사용하였다.

본 논문에서는 크게 세 종류의 실험을 진행하였다. 첫 번째 실험에서는 ITR을 변화하면서 최적의 ITR을 찾았고, 두 번째 실험에서는 최적의 ITR을 적용한 상황에서 본 논문에서 제안한 두 가지의 자료구조 재사용 기법을 적용하여 얻을 수 있는 성능 개선 효과를 검증하였으며, 마지막 실험에서는 본 논문에서 새로 제안하는 방법과 기존의 세가지 방법을 통합 적용하여 얻을 수 있는 성능 개선 효과를 검증하였다.

4.2 ITR 값에 따른 실험 및 성능 분석

본 논문에서 소개하는 멀티코어에서의 성능 개선 방법, 대용량 패킷 처리시스템에서 성능 개선 방법, sk_buff recycling, fd 재사용, epoll_event 재사용을 통합 적용한 상태에서 표 4와 같이 ITR 값을 변경하면서 성능 비교 실험을 수행하였다. 서버 애플리케이션은 간단한 웹 서버를 이용하여 측정 비교하였다. 클라이언트에서 아파치벤치를 이용하여 진행하였고, 각각의 실험은 ITR 값을 바꿔가며 CPS(Connection Per Second)을 측정하였다. 데이터 크기는 1k로 진행하였다.

우선 그림 12를 보면 ITR 값이 1000에서 커질수록 성능이 감소하는 것을 볼 수 있다. Default의 경우 Dynamic 동작 방식으로 ITR값이 1000에서 400000 사이의 값으로 수시로 변경되면서 동작하는 방법이다. ITR의 값이 1000일 경우 Default 값에 비교하여 약 15% 성능 증가하였으며 ITR의 값이 400000의 경우 Default 값에 비교하여 약 16% 떨어졌다. ITR을 OFF 상태로 하면(=0) Default 값에 비교하여 약 21% 성능이 떨어졌다. ITR 값이 1000일 경우 CPS가 가장 높은 것을 볼 수 있다.

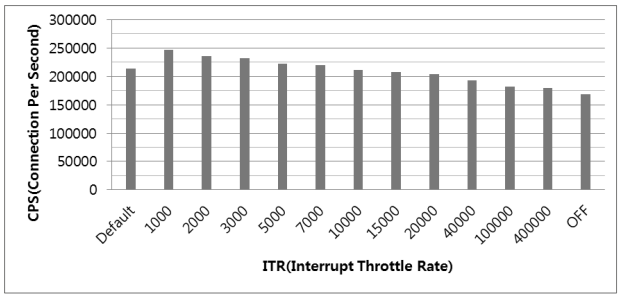

Fig. 12. Experiment Result of CPS with simple web server

4.3 본 논문에서 추가 제안하는 자료구조 재사용 실험 및 성능 분석

본 실험은 표 4와 같이 진행되었다. 실험1은 4.2에서 성능이 가장 좋은 ITR 1000값을 기본 서버에서 적용하여 측정하였고, 실험2는 sock 자료구조체 재사용을 추가하고, 실험3의 경우 sock 자료구조와, epoll_event 자료구조 재사용을 추가하여 실험을 진행하였다.

Table 4. Experiment Procedure

실험1	기본 서버 + ITR(1000)
실험2	실험1 + fd 재사용 적용
실험3	실험2 + epoll_event 재사용 적용

서버애플리케이션은 간단한 웹 서버를 이용하여 측정 비교하였다. 클라이언트에서 아파치벤치를 이용하여 실험1부터 실험3까지 진행하였고, 각각의 실험에서 CPS(Connection Per Second)를 측정하였다. 데이터 크기는 1k로 진행하였다.

그림 13을 보면 실험1에서 3으로 가면서 성능이 향상됨을 알 수 있다. 실험2에서 fd 재사용 적용하였을 때 실험1 대비 약 7% 성능이 향상되었으며, 실험3에서 epoll_event 재사용 적용하였을 때 실험2 대비 약 6% 성능이 향상되었다.

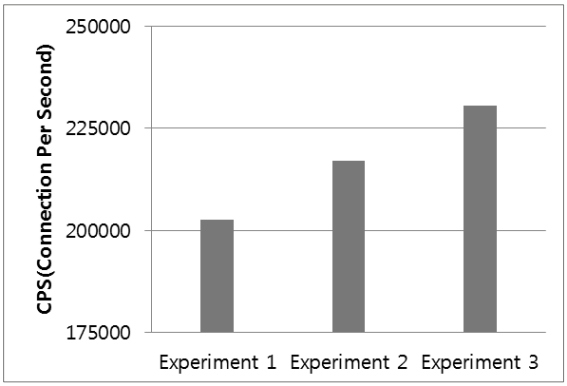

Fig. 13. Experiment Result of CPS with simple web server

4.4 제안된 통합 적용을 통한 실험 및 성능 분석

실험은 표 5과 같이 진행되었다. 각각의 실험은 이전 실험에서 한 가지 방법을 추가 적용하였고 서버애플리케이션을 바꿔가며 실험1부터 실험6까지를 반복했다.

Table 5. Experiment Procedure

실험1	기본 서버
실험2	실험1 + 멀티코어에서의 성능 개선 방법 적용
실험3	실험2 + 대용량 패킷 처리시스템에서 성능 개선 방법 적용
실험4	실험3 + sk_buff recycling 적용
실험5	실험4 + fd 재사용 적용
실험6	실험5 + epoll_event 재사용 적용

서버애플리케이션은 간단한 웹 서버, Lighttpd, Apache 순으로 진행하였다. 클라이언트에서 아파치벤치를 이용하여 실험1부터 실험6까지 진행하였고, 각각의 실험은 데이터 요청 크기를 바꿔가며 CPS(Connection Per Second)와 처리량(throughput)을 측정하였다. ITR 설정은 4.2의 실험결과를 토대로, CPS 성능이 가장 좋은 1000을 사용하였다.

그림 14를 보면 단순한 웹 서버에서는 실험마다 성능이 향상됨을 알 수 있다. 그렇지만, 데이터 크기가 커짐에 따라

성능 향상 폭이 감소하다가 결국 실험1부터 실험6까지 모든 실험들의 CPS가 같아지는 것을 볼 수 있었다. CPS가 동일해지는 부분은 그림 15를 통해 알 수 있었다. 그림 15에서 데이터 크기가 커짐에 따라 처리량이 증가하다가 9.4Gbps에서 유지되는 것을 볼 수 있다. 9.4Gbps는 실험에서 사용하는 NIC가 낼 수 있는 최고 성능으로 보이며, CPS가 동일해지는 현상은 대역폭의 한계로 볼 수 있다.

그림 16과 그림 17을 보면, Lighttpd를 사용한 실험결과가 위의 단순한 웹 서버를 사용한 실험결과와 비슷함을 알 수 있다. 데이터 크기가 가장 작을 때 가장 크게 성능이 향상되

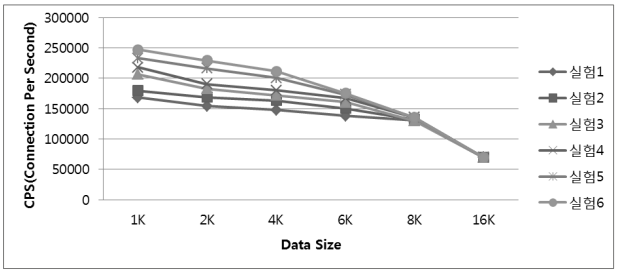

Fig. 14. Experiment Result of CPS with simple web server

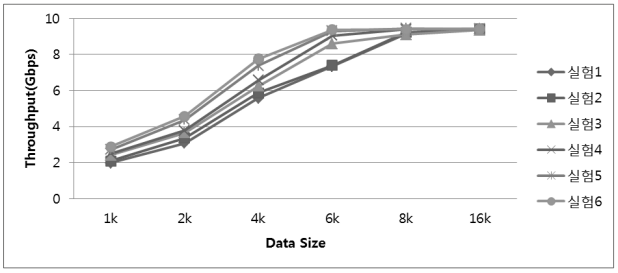

Fig. 15. Experiment Result of Throughput with simple web server

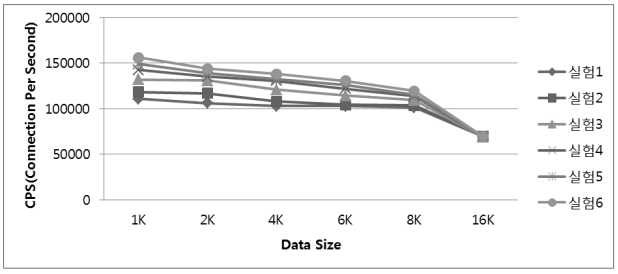

Fig. 16. Experiment Result of CPS with Lighttpd

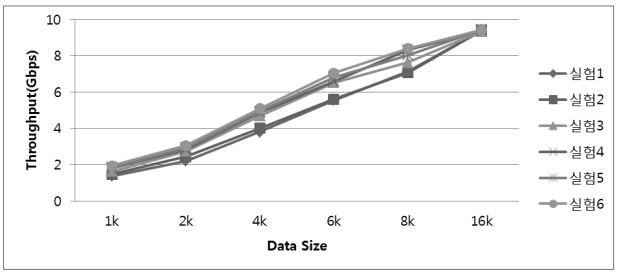

Fig. 17. Experiment Result of Throughput with Lighttpd

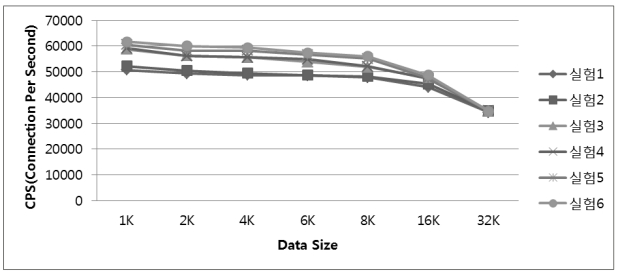

Fig. 18. Experiment Result of CPS with Apache

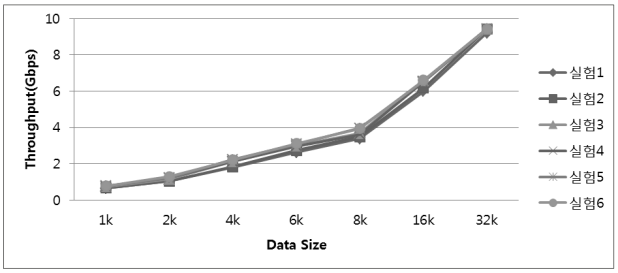

Fig. 19. Experiment Result of Throughput with Apache

었으며, 데이터 크기가 커지면서 성능 향상 정도가 줄어든다. 그림 17에서 데이터 크기 16k에서 처리량이 9.4Gbps에 도달하고 이때 그림 16의 CPS는 동일해지는 것을 볼 수 있다.

마지막으로 그림 18과 그림 19를 보면, Apache를 사용한 실험 결과가 앞선 다른 두 종류의 서버를 이용한 실험 결과와 비슷하지만 전체적으로 성능 향상 정도가 낮은 것을 볼 수 있다. 간단한 웹 서버와 Lighttpd를 사용하는 경우 싱글 스레드로 동작하지만, Apache를 사용하는 경우 멀티스레드로 동작한다. 이러한 구조상의 차이가 성능의 차이를 만드는 것으로 보인다. Lighttpd가 동작할 때 초당 7500번의 문맥교환이 발생하는 반면, Apache의 경우 초당 21만번의 문맥 교환이 발생하여 Lighttpd에 비해 28배 오버헤드가 높다는 것을 볼 수 있다.

4.5 토론

첫 번째 실험에서 ITR 값에 따른 성능 향상을 검증하였으며, ITR 값이 1000일 경우 CPS가 가장 높은 것을 볼 수 있었다. 기본적으로 default 값은 dynamic으로 설정되어 있으며 ITR 값이 1000에서 400000 사이를 매번 변화한다. ITR 값에 따른 기본적인 내용은 표 2를 통해 살펴볼 수 있으며, ITR 값이 작을수록 Latency는 증가하지만 CPU의 부하는 줄어들게 된다. 그렇기 때문에 만약 Latency에 많이 민감한 경우에는 높은 ITR값을 사용하는 것이 좋을 수 있다. 하지만 높은 부하 상황에서는 잦은 인터럽트로 인하여 CPU 부하를 초래하며 ITR이 낮을 때 Latency가 더 좋아지는 역전 현상이 발생할 수 있다. 낮은 부하에서는 ITR 값이 높을 때, 그리고 높은 부하에서는 ITR 값이 낮을 때 Latency가 낮아지고 CPS가 높아지는 것을 볼 수 있다. 본 논문에서는 CPU를 100%로 사용할 때의 CPS를 기준으로 실험했다. 높은 부하 상황으로 ITR 값을 낮춰 더 좋은 CPS를 얻을 수 있었다.

두 번째 실험에서 본 논문에서 추가 제안하는 자료구조 재사용의 성능 향상을 검증하였다. 간단한 웹 서버 기준으로 CPS를 측정하였으며, fd 재사용에서는 약 7%, epoll_event 재사용에서는 약 6%, 총 13% 성능이 향상됨을 확인하였다.

세 번째 실험에서 기존 세 가지의 방법과 본 논문에서 추가 제안한 두 가지의 방법을 통합 사용하여 성능 향상을 검증하였다 간단한 웹 서버를 사용한 경우 CPS가 최대 40% 성능 개선효과를 보였으며, Lighttpd와 Apache 웹 서버를 사용한 경우 각각 36%, 20% 성능 개선 효과를 보였다.

본 논문의 실험 결과를 기존 연구 결과들과 비교해보면 다음과 같다. 우선 표 1에서 멀티코어 환경에서의 흐름 제어에 관한 기존 연구[2]를 살펴보게 되면 300% 성능 향상이 보고되었지만 본 실험 과정에서는 약 6% 성능이 향상되어 그 차이가 큰 것을 볼 수 있다. 기존 연구[2]의 경우에는 한 번에 하나의 코어에서만 TCP 스택 처리과정을 할 수 있던 것을 본 연구에서는 코어당 하나의 NIC를 할당하여 동시에 다수의 코어들이 TCP 스택을 처리할 수 있게 하였다. 본 연구에서는 추가적으로 core affinity 적용하여 2단계 성능 개선효과가 있었다. 싱글코어만 사용하여 처리하던 경우 대비 300% 성능 향상이 되었음을 의미한다. 하지만 본 논문의 실험에서 사용한 NIC 모델은 하나의 NIC에서 멀티 큐를 지원한다. 코어당 하나의 큐를 할당하고 동시에 다수의 코어가 패킷 수신을 할 수 있게 구현되어 있다. 최근 이더넷 드라이버에서는 이 기능을 기본적으로 지원하고 있다. 본 연구에서는 이 기본 기능을 사용하는 환경에서 core affinity를 추가로 적용하여 약 6%의 성능 향상을 이끌어냈다는 것을 의미한다.

sk_buff 자료구조 재사용에 관한 기존 연구[10]의 실험 결과를 살펴보면 sk_buff 재사용을 통해 약 6% 성능이 향상되었다. 본 논문에서는 그림 14에서 실험3과 4를 비교해 보면 sk_buff 재사용을 통해 약 5% 성능이 향상되었는데 기존 연구의 실험 결과와 비슷하다고 하겠다.

본 논문에서 사용한 메모리 재사용을 위해서는 애플리케이션의 수정이 필요했다. 메모리를 초기화하는 부분과 재사용을 위한 부분인데, 3장에서 설명한 것과 같이 기존 API를 이용하여 수정을 최소화하였다.

5. 결 론

본 논문에서는 기존 TCP 네트워킹 수신과정에서 발생하는 성능 저하 문제점들을 파악하고 이를 해결할 수 있는 방법을 제안하였다. 리눅스 TCP 네트워킹 성능 개선에 관한 기존 방법 세 가지와 본 논문에서 새로 제안하는 두 가지 방법을 통합 적용하여 성능을 향상시켰다. 기존 연구들은 하나의 성능 저하 요소를 부각시키고 개선 방법을 제안하고 있다. 기존 개선방법들 중에 동시 적용이 가능한 부분이 있다면 통합 적용이 더 큰 성능 향상을 기대할 수 있다는 가정에 출발하였다.

본 논문에서 사용한 기존 개선 방법들로는 멀티코어 환경에서 패킷을 흐름 단위로 코어에 할당하는 방법, 과도한 인터럽트 요청을 조절하는 ITR(Interrupt Throttle Rate) 방법, sk_buff 자료구조 recycling방법이다. 본 논문에서 새로 제안하는 방법은 fd 자료구조 recycling 방법과 epoll_event 자료구조 recycling 방법이다.

웹서버 환경에서 실험을 통해 본 논문의 제안방법들의 성능 개선 효과, 또한 기존방법들과의 통합 적용했을 경우 성능 개선 효과를 검증하였다. 웹 서버로는 간단한 웹 서버, 리눅스에서 일반적으로 사용하는 Lighttpd와 Apache 웹 서버를 사용하였다. 간단한 웹 서버를 사용했을 경우 본 논문에서 제안한 fd 재사용과 epoll_event 재사용을 적용할 경우 성능이 각각 7%와 6% 개선되었고, 기존 방법 세 가지와 더불어 총 다섯 가지 방법을 통합 적용한 경우 성능이 총 40% 까지 개선되었다. Lighttpd와 Apache 웹 서버 환경에서 다섯 가지 통합 방법을 적용한 경우 성능이 각각 총 36%, 20%까지 개선되었다.

Reference

[1] Wenji Wu, Phil DeMar, and Matt Crawford, “A Transport-Friendly NIC for Multicore / Multiprocessor Systems”, IEEE Transaction on Parallel and Distributed Systems, Vol.23, No.4, pp.607-615, Apr., 2012.

[2] H. Kwon, H. Jung, H. Kwak, K. Chung, and Y. Kim, “Performance Improvement of Linux TCP Networking based on Flow-Level Parallelism in a Multi-Core System”, The KIPS Transaction: Part A, Vol.16, No.2, pp.113-124, 2011.

[3] S. J. Baek and J. M. Choi, “Internal structure of the Linux kernel”, Gyohaksa, Korea, pp.230-231, 2008.

[4] Intel, “Interrupt Moderation Using Intel Gigabit Ethernet Controllers”, April 2007, [Internet], http://www.intel.com/content/dam/doc/application-note/gbe-controllers-interrupt-moderation-appl-note.pdf

[5] Intel whitepaper, “Supra-linear Packet Processing Performance With Intel Multi-core Processors”, 2006, [Internet], http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/multicore-supra-linear-packet-processing-paper.html

[6] Intel Executive Summary, “Accelerating Security Applications With Intel Multi-core Processors”, 2006, [Internet], http://www.intel.in/content/dam/www/public/us/en/documents/technology-briefs/multi-core-processors-security-brief.pdf

[7] E. Lemoine, C. Phamand, and L. Lefèvre, “Packet Classification in the NIC for Improved SMP-based Internet Servers”, IEEE Proceedings of the International Conference on Networking(ICN 2004), Guadeloupe, French Caribbean, Feb., 2004.

[8] Linux Base Driver for the Intel(R) Ethernet 10 Gigabit PCI Express Family of Adapters, [Internet], http://downloadmirror.intel.com/14687/eng/readme.txt

[9] C. Walravens, “Receive Descriptor Recycling for Small Packet High Speed Ethernet Traffic”, Electrotechnical Conference, MELECON 2006, IEEE Mediterranean, pp.1252-1256, 2006.

[10] R. Bolla and R. Bruschi, “A high-end Linux based Open Router for IP QoS networks: tuning and performance analysis with internal (profiling) and external measurement tools of the packet forwarding capabilities”, Proc. of the 3rd International Workshop on Internet Performance, Simulation, Monitoring and Measurements(IPS MoMe 2005), Warsaw, Poland, Mar., 2005.

[11] apachebench, [Internet], http://httpd.apache.org/docs/2.0/programs/ab.html

[12] http://www.kegel.com/c10k.html

[13] Raffaele Bolla and Roberto Bruschi, “The IP Lookup Mechanism in a Linux Software Router: Performance Evaluation and Optimizations”, Proc. of the 2007 IEEE Workshop on High Performance Switching and Routing.

[14] Open Router resources from the TNT Lab. [Internet], www.tnt.dist.unige.it

[15] R. Bolla and R. Bruschi, “RFC 2544 Performance Evaluation and Internal Measurements for a Linux Based Open Router”, Proc. of IEEE 2006 Workshop on High Performance Switching and Routing(HPSR 2006), Poznan, Poland, pp. 9-14, June, 2006.

[16] Seokoo Kim and Kyusik Chung, “A Performance Improvement of Linux TCP Networking by Data Structure Reuse”, Proc. of 2013 KICS(Korea Information and Communications Society) Fall Conference, Session 9A-1, pp.251-252, Nov., 2013.

김 석 구

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

2011년~2013년 펌킨 네트웍스/개발 엔지니어

2012년 숭실대학교 정보통신전자공학부 (학사)

2014년 숭실대학교 정보통신공학과(석사)

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