프로그램이 가능한 기가빗 네트웍 인터페이스 카드 상에서의 네트웍 스택 성능 측정

이 승 윤, 박 규 호
카이스트 전기 및 전자공학부
e-mail : sylee@core.kaist.ac.kr

Performance Evaluation of network stack with programmable Gigabit Network Interface Card

Seung-Yoon Lee, Kyu-Ho Park
Department of Electrical Engineering & Computer Science Division of Electrical Engineering
Korea Advanced Institute of Sciencte and Technology

Abstract

Ethernet is one of the most successful LAN technologies. Now gigabit ethernet is available in real network and some network interface cards(NIC) supports TCP segment offloading (TSO), IP checksum offloading(ICO), Jumbo frame and interrupt moderation. If we use this features appropriately, we obtain high throughput with low CPU utilization. This paper represents the network performance by varying above features.

I 서론

네트웍에서의 전송속도가 높아짐에 따라 어플리케이션(application)까지 이러한 속도 증가를 적용하기가 점점 더 어려워지고 있다. 이러한 대역폭(bandwidth)를 어플리케이션까지 적용하려면 I/O 버스의 속도, 메모리 카피가 일어나는 비율, 프로세서의 성능 등과 같은 여러 곳에서 버틀넥(bottleneck)이 생기게 된다. 본 논문에서는 프로세서의 성능에 관련된 버틀넥을 개선시키려 할 것이다. 특히 본 논문에서는 통신 프로토콜(protocol)에 관련된 연산을 호스트(host) 프로세서가 아닌 프로그램이 가능한 네트웍 인터페이스 카드(Network Interface Card)가 하도록 할 것이다. 이러한 프로토콜의 오프로딩(Offloading)은 두 가지의 중요한 장점을 가지고 있다. 첫째로 프로세서가 통신 프로토콜에 대한 연산을 하지 않게 되어 프로세서에 의한 버틀넥을 없앨 수 있고 이는 네트웍 대역폭의 향상을 가져올 수 있다. 둘째로 통신 프로토콜에 대한 연산을 네트웍 인터페이스 카드가 하게 되어 호스트의 프로세서는 사용도가 떨어지게 되고 더 많은 시간을 다른 어플리케이션 프로그램을 위해 사용할 수가 있다.

프로토콜 연산을 오프로딩하는 것은 분명한 장점이 있지만 네트웍 인터페이스 카드가 얼만큼의 연산을 담당하느냐는 신중히 선택해야 한다. 하지만 요즘의 네트웍 인터페이스 카드는 기가빗 이더넷(gigabit ethernet)을 지원하기에 충분한 연산 능력을 가지고 있다. 다음 절에서는 TCP/IP 오프로딩과 인터럽트의 수를 줄이는 방법을 설명하고 3절에서는 실험의 결과를 설명하도록 하겠다. 마지막으로 4절에서는 실험의 결론과 향후 연구 방향에 대해서 설명하겠다.

II 네트웍 인터페이스 카드의 성능

2.1 TCP/IP 오프로딩

데이타 통신을 할 때 전송하는 측에서의 오프로딩과 수신하는 측에서의 오프로딩을 생각해 볼 수 있다.

전송하는 측에서의 오프로딩을 살펴보면 아래의 [그림1]과 같다. 더 높은 계층에서 페이로드(payload)를 받은 뒤에 이를 TCP/IP로 감싸지게 된다. 오프로딩을 수행하는 모듈(네트웍 인터페이스 카드)은 데이타그램(datagram)의 길이, fragmentation, identification,

TTL(Time To Live), Flags, checksum과 같은 헤더의 요소를 채워넣어야 할 것이다. 전송하는 측에서는 버려지는 데이타그램이 없다는 것이 수신단에서의 오프로딩보다 더 간단한 점이라 할 수 있다.

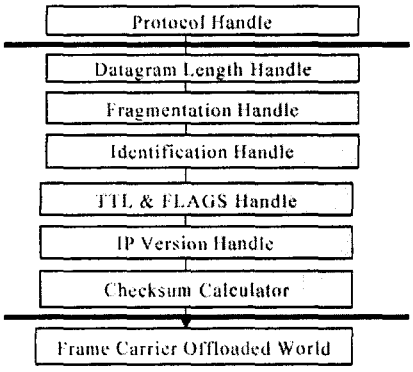

그림1. 전송하는 측에서의 TCP/IP 오프로딩

데이터를 수신하는 측에서는 데이타 링크(Data link)계층에서 프레임이 오류없이 전송된 다음에 프로토콜에 관련된 연산이 가능하다. 수신하는 측에서는 처음에 IP의 버전부터 체크해야 할 것이다. IP 헤더의 구성은 버전이 IPV4인지 IPV6인지에 따라 다르기 때문이며 버전에 대한 정보가 일치한다면 IP 헤더의 정보가 맞는지 확인해야 한다. 그 다음에는 IP 패킷의 idetificator를 확인하고 fragment에 대한 정보를 확인하여 정보가 일치하다면 상위 계층으로 데이타를 넘겨주게 될 것이다. 하지만 에러가 생긴다면 best effort의 접근방법인 IP에서는 에러를 핸들(handle)하지 않고 수신된 패킷을 그냥 버리게 된다. 이와 같은 것을 그림으로 표현하면 아래 그림과 같다.

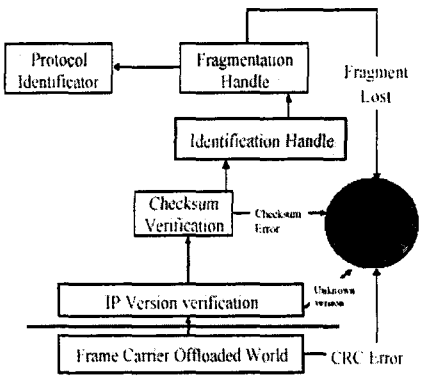

그림2. 수신하는 측에서의 TCP/IP 오프로딩

2.2 Jumbo frame and Interrupt moderation(IM)

인터럽트에 의한 오버헤드를 살펴보면, 기가빗 이더넷 상황에서 1500바이트의 프레임을 전송한다고 가정하자. 또한 전송하는 측에서 수신하는 측까지의 전송시간은 거의 없다고 가정했을 때

$$1500B \times \frac{8b}{B} \times \frac{1}{1000Mb/s} = 12us$$

기본적으로 12us마다 호스트의 CPU에 인터럽트를 생성하게 된다.

데이터 통신의 오버헤드를 줄이기 위해서는 2가지 방법을 생각해 볼 수 있다. 점보 프레임을 사용하는 것과 IM을 사용하는 것이다. 점보 프레임은 프레임 사이즈를 1500바이트 이상으로 하여 인터럽트가 더 늦게 생성되게 한다. 하지만 이 방법은 사이즈가 큰 메세지에서 사용이 가능하며 스위치가 점보 프레임을 지원해 주어야 한다.

IM의 경우에는 일정한 수의 패킷이 도착하거나 일정한 시간이 지났을 경우에 CPU에 인터럽트를 생성하는 것이다. 만약 한 노드에 일정한 개수의 프레임이 도착한다고 가정하면 IM은 인터럽트를 여러개 모아서 CPU에 전달하므로 인터럽트의 수를 크게 줄일 수 있을 것이다. 하지만 이러한 방법은 높은 속도의 네트웍에서는 가능하지만 작은 메세지에 대한 latency가 커지게 되는 단점이 있다.

Ⅲ 실험결과

이번 실험에서 netperf를 벤치마크 프로그램으로 사용하였다. 2개의 컴퓨터를 사용하였으며 각각은 2.8GHz Pentium4, 1Gbyte 메모리 그리고 Intel PRO/1000 MT Server Adapter를 장착하였으며 두 개의 컴퓨터는 크로스 케이블로 연결하였다.

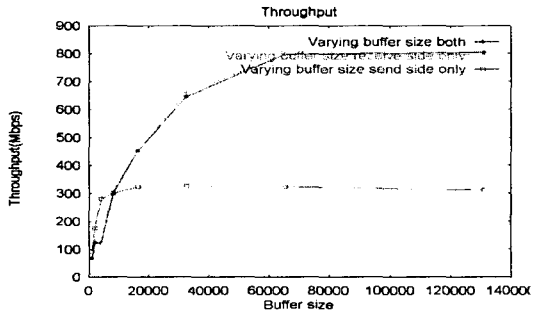

그림3. 버퍼 사이즈를 변경했을 때의 throughput

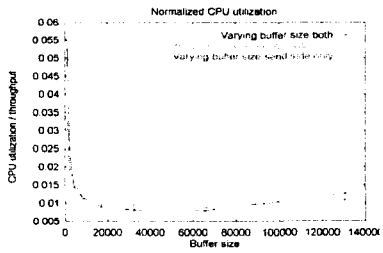

그림4. 버퍼 사이즈를 변경했을 때의 CPU사용도

위 그림을 보면 버퍼 사이즈가 증가하는데 비례해서 throughput이 증가하는 것을 볼 수가 있다. 그러나 전송하는 측에서만 버퍼 사이즈를 증가하고 수신하는 측에서는 버퍼 사이즈를 고정한다면 네트웍은 300Mbps에서 포화됨을 알 수가 있다. 이는 전송하는 측에서 많은 데이타를 수신하는 측에 전송해 주어도 수신하는 측에서 이를 처리하지 못하기 때문에 발생하는 문제이다. 이는 수신하는 측에서만 버퍼 사이즈를 증가시키면 수신하는 측과 전송하는 측 모두의 버퍼 사이즈를 증가시킬 때와 같은 성능이 나오는 것을 보아도 확인할 수가 있다. CPU 사용도를 보면 버퍼 사이즈가 작을 때는 CPU를 많이 사용하다가 버퍼 사이즈가 커지면 급격히 사용도가 줄어드는 것을 볼 수가 있다. 이는 기본적으로 TSO, ICO를 사용하기 때문에 네트웍 속도의 증가가 CPU 사용으로 이어지지 않기 때문이다. 버퍼 사이즈가 일정 크기 이상으로 증가하면 네트웍은 포화되어 더이상 속도가 증가하지 않는 것을 알 수 있다. 하지만 이때 CPU 사용도가 증가함을 알 수가 있다. 결국 버퍼 사이즈를 늘리는 것만이 좋은 것이 아니라 주어진 시스템에 따라 적절한 버퍼 사이즈의 값이 있다는 것을 알 수가 있다. 우리가 실험한 시스템에서는 64Kbyte정도의 버퍼 사이즈가 적당한 것으로 나타났다.

4.1 TSO and ICO

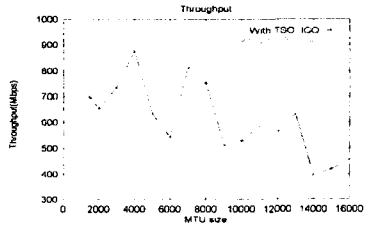

그림5. TSO, ICO에 대한 throughput

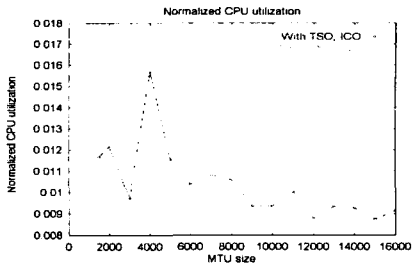

그림 6. TSO, ICO에 대한 CPU 사용도

	Average RTT (data size = 50000bytes)
With TSO, ICO	1.539ms
Without TSO, ICO	1.629ms

실험결과를 보면 TSO, ICO를 사용하지 않았을 때 throughput이 더 높아지는 것을 알 수가 있다. 하지만 CPU 사용도는 TSO, ICO를 사용할 때 훨씬 좋은 것을 알 수가 있다. CPU 사용도의 절대적인 값을 보면 TSO, ICO를 사용할 때는 6%정도의 CPU 사용도를 가지고 있었고 사용하지 않았을 때는 10% 이상의 CPU 사용도를 가지고 있었다. 이를 좀더 확실히 하기 위해 ping을 사용하여 데이타 크기를 50,000바이트로 패킷을 보내보았다. TSO, ICO를 사용하지 않았을 경우에 좀 더 작은 latency를 보였는데 이를 보면 네트웍 인터페이스 카드가 모든 것을 처리하는 것보다는 CPU 사용도가 낮을 때는 CPU가 모든 것을 처리하는 것이 더 빠를 수 있다는 것을 보여주고 있다. 우리가 수행한 실험에서는 CPU 사용도가 6%에서 15% 정도였기 때문에 이러한 결과가 나온 것 같다. 하지만 SpecWeb99를 사용했을 때는 다른 결과가 나타났다. 아래 그림7을 보면 TSO, ICO를 사용했을 때 성능이 30% 정도 향상된 것을 볼 수가 있는데 이 때는 디스크 오퍼레이션이 있어서 CPU 사용도가 항상 100%였다. 결국 호스트의 CPU 사용도가 낮을 때는 TSO, ICO를 사용하지 않는 것이 좋고 CPU 사용도가 높을 때는 연산을 분산하는 것이 더 좋은 성능을 보인다는 것을 알 수가 있었다.

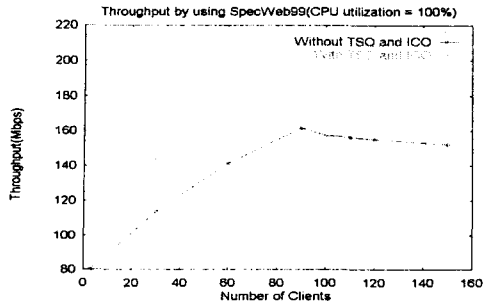

그림7. SpecWeb99를 사용했을 때의 throughput

4.2 Interrupt Moderation

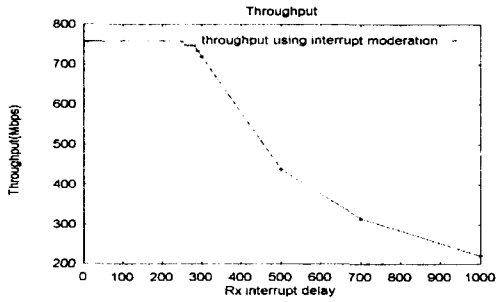

그림8. IM을 사용했을 때의 throught

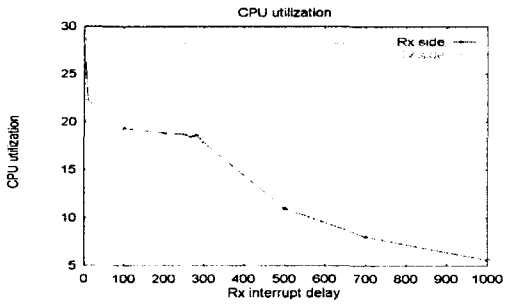

그림9. IM을 사용했을 때의 CPU 사용도

우리는 수신하는 측에서 적당한 수의 인터럽트를 모아서 CPU에게 전달해 주면 네트워의 속도는 유지하면서 CPU 사용도는 줄어들 것이라고 예상하였으며 실험결과도 그와 같았다. 아래 그림 8, 9를 보면 인터럽트 지연을 250us까지 하여도 네트웍 속도는 그대로 유지가 되었으며 CPU 사용도는 30% 이상 줄어든 것을 알 수가 있다. 하지만 더욱 재미있는 결과는 인터럽트 지연을 100us로 할 때와 250us로 할 때의 CPU 사용도는 비슷한 것으로 나타났다. 결국 크기가 작은 메세지의 경우에는 인터럽트 딜레이가 커질수록 latency가 커지므로 모든 것을 고려했을 때 인터럽트 지연을 100us 정도로 할 때 가장 좋은 성능을 나타낸다고 볼 수 있겠다.

Ⅳ 결론 및 향후 연구방향

이번 실험에서는 리눅스 커널 2.6에서의 새로운 기능인 TSO, ICO 등을 이용하여 기가빗 이더넷 상황에서 네트웍의 속도는 유지하면서 CPU의 사용도를 줄이는 데 대한 여러 실험을 하였다. zero-copy 방법을 도입한다면 네트웍 속도를 유지하면서 CPU 사용도를 줄일 수 있을 것이다. 일대일로 연결된 시스템이 아닌 서버, 클라이언트 모델로 하여 접속이 많을 때는 CPU 사용도가 높아질 것이며 이때 통신 프로토콜을 위한 연산을 CPU와 네트웍 인터페이스 카드에 분산시킨다면 기존의 서버보다 더 많은 접속을 허용할 수 있을 것이다. 앞으로는 이에 관련된 load-balancing 서버에 관한 연구가 필요해질 것이다.

참고문헌

[1] Netperf manual
[2] SpecWeb99 manual
[3] Maccabe, A.B., Experience in offloading protocol processing to a programmable NIC. Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on, 23-26 Sept. 2002
[4] Juan M. Isidoro C. A TCP/IP framework for TCP/IP offloading implementation. CRC 2002