제 1 장

서론

1969년 미국에서 ARPANET[21] 이 등장한 이래 많은 종류의 데이터 통신망이 구축되었고, 이제는 모든 컴퓨터가 어떤 형태이든 간에 통신망에 연결되어 사용되는 시대가 되었다[33]. 더구나 1982년 TCP/IP(Transmission Control Protocol/Internet Protocol)를 이용해 인터네트를 구축함에 따라 이제는 "정보의 바다"라는 말이 생겨났고, 인터네트를 사용하여 필요한 정보는 무엇이든 얻을 수 있는 시대가 되고 있다. 인터네트가 처리하는 정보의 형태도 숫자나 문자에서 점차 음성이나 화상 정보를 다루는 멀티미디어 쪽으로 발전하고 있다. 특히, 월드 와이드 웹(World Wide Web) 서비스의 출현으로 이러한 추세는 계속될 것으로 보인다[3].

현재의 통신 시스템의 근간인 인터네트는 독립적인 4 개의 계층으로 나누어져 있다[33]. 이들 계층은 응용 계층, 트랜스포트 계층, 네트워크 계층, 데이터링크 계층이며, 각 계층은 하나 또는 몇 개의 프로토콜 및 구현 기술로 구성되어 있다. 응용 계층은 인터네트 사용자를 위한 인터페이스를 제공하는 부분 또는 인터네트의 프로토콜을 사용하는 응용 프로그램들이라고 할 수 있다. 트랜스포트 계층은 각 호스트간(end-to-end)의 데이터 흐름을 관장하는 부분이다. 이 계층에서 많이 쓰이는 프로토콜은 TCP[31], UDP(User Datagram Protocol)가 있다. TCP

는 연결 지향(connection-oriented) 프로토콜이며 호스트간에 신뢰성 있는 데이터 전송이 가능하도록 해준다. 즉, 통신망의 트래픽(traffic) 특성에 따라 데이터 전송률을 조절할 수 있고, 에러에 대한 대처를 할 수가 있다. 반면, UDP는 비연결형(connectionless) 프로토콜이며 사용자의 데이터를 데이터그램(datagram)이라고 하는 형태로 일시에 전송하는 특징을 가진다. 에러 및 트래픽 특성에 대한 고려를 하지 않는 관계로, UDP를 사용하는 시스템일 경우 이러한 제어는 응용 계층이 담당하게 된다. 네트워크 계층은 네트워크 상의 데이터 또는 패킷을 적절한 네트워크 또는 호스트로 전달(switching)하는 기능을 한다. 인터네트의 경우, 네트워크 계층의 기능은 IP[30]가 담당을 하고 이를 제어하기 위한 ICMP(Internet Control Message Protocol), IGMP(Internet Group Management Protocol) 등의 프로토콜이 존재하게 된다. 데이터링크 계층은 데이터가 전송되는 물리적인 매개체를 의미한다. 데이터링크 계층에 많이 사용되는 기술로는 이더넷(Ethernet)[8], 토큰링(Token Ring), 비동기 전송 방식(Asynchronous Transfer Mode)[25] 등이 있다. 따라서, 인터네트가 멀티미디어 데이터를 고속으로 처리할 수 있을 지의 여부는 각 계층의 처리 능력에 달려 있다고 할 수 있다. 데이터링크 계층에 있어서는 광케이블 및 ATM이 등장하여 대량의 데이터를 처리하기 위한 기틀이 마련되었고, 이 부분은 더 이상 통신 시스템의 병목 부분이 되지 않고 있다. 따라서 초고속 통신 시스템을 구현하기 위해 필요한 조건은 고속의 네트워크 계층 및 트랜스포트 계층을 구현하는 것이 될 것이다[29, 28]. 특히, 신뢰성 있는 전송을 가능하게 해주는 고속의 TCP/IP를 구현하는 것이 문제의 핵심이라고 할 수 있겠다.

고속의 TCP/IP를 구현하기 위한 연구의 전 단계로서 기존의 TCP/IP 구현에 대한 성능을 분석[23, 9, 5, 14, 27, 18, 15]하여 병목 부분을 찾아내려는 노력이 계속되었다. 또한 개선된 TCP/IP 구현에 대해 그 향상도를 측정하기 위한 성능

분석[13, 17]도 수행되었다. 따라서 성능 분석은 주어진 시스템에 대해 성능 향상을 위한 방향을 제시하는 중요한 과정이라 하겠다. 초기의 성능 분석에 관한 연구는 TCP/IP 프로토콜 자체에 국한되어 있었지만, 최근의 분석 결과들은 TCP/IP의 성능이 운영체제와의 상호 작용과 밀접한 관계가 있다는 사실을 말해 주고 있다[14]. 현재까지 알려진 TCP/IP의 병목 부분은 데이터 복사, 체크섬 계산, 메모리 관리, 타이머 관리 등이며[5] 이에 대한 소프트웨어적인 측면[13]에서, 혹은 구조적인 측면[20, 19]에서의 개선 방향이 제안되었고, 그에 따른 구현[16]과 성능 분석[17]이 수행되었다.

지금까지의 분석 결과들은 기존의 TCP/IP 구현과 개선된 구현을 비교/분석하여 상대적인 성능(throughput) 향상에 대한 비율을 제시하였다. 그러나 고성능 통신을 위해서는 TCP/IP의 절대 성능을 알아내는 것이 중요하다 하겠다. 다시 말해, TCP/IP가 고속 및 광대역의 통신 환경에서 어느 정도의 성능을 발휘할 것인지를 예측하는 것이다. 본 논문에서는 TCP/IP의 데이터 전송/수신 성능에 초점을 맞추어 병목 부분을 정량적으로 분석하였다. 즉, 연결이 설정된 이후의 데이터 전송/수신 및, ACK(acknowledgment)처리 과정을 분석하였다. 성능 분석 요소는 TCP/IP 수행 시의 싸이클 수, 수행 명령어 수, 메모리 참조 등이다. 이를 통해 TCP/IP의 수행 특성을 살펴보았고 프로그램의 복잡도, 메모리 대역폭을 측정하여 TCP/IP 처리 능력의 상한선을 살펴보았다.

본 논문의 구성은, 2장에서 TCP/IP 동작을 개략적으로 설명하고, 3장에서는 현재까지 알려진 컴퓨터 시스템의 성능 분석 방법을 비교/분석하고 본 논문에서 사용한 성능 분석 방법을 설명한다. 4장에서는 TCP/IP의 전송/수신 성능 측정 환경 및 방법에 대해 기술하고, 성능 평가 기준을 제시한다. 5장에서는 실험 결과에 대한 분석을 통해 TCP/IP의 전송/수신 능력의 상한선을 제시하며, 6장에서 결론

을 맺는다.

제 2 장

TCP/IP 프로토콜 및 관련 프로토콜의 개요

TCP/IP는 인터네트 상에서 데이터를 신뢰성 있게 전송/수신할 수 있는 프로토콜이나 독자적으로 동작할 수는 없으며, 데이터링크 계층으로부터, 또는 사용자 계층으로 적절한 기능을 제공받거나 제공해야 한다. TCP/IP 및 연관 프로토콜들의 계층 관계와 각 프로토콜간에 일어날 수 있는 자료 흐름을 그림 2.1에 도시하였다. 데이터 전송/수신 시의 일반적인 자료 흐름은 응용 프로그램, 트랜스포트 프로토콜(TCP/UDP), 네트워크 프로토콜, 데이터링크 프로토콜이 개입되어 일어난다. 그러나 응용 프로그램이 데이터 전송이 아닌 네트워크의 상태 점검 및 감시를 위해 작성될 경우 트랜스포트 프로토콜이 없이도 통신을 할 수 있다. TCP/IP를 이용하는 데이터 통신은 직접적으로 응용 프로그램, TCP[31], IP[30], 데이터링크 프로토콜을 통해 수행되며, 간접적으로는 ARP, RARP, ICMP, IGMP 등으로부터 서비스를 받아 수행된다.

일반적으로 여러 개의 프로토콜로 구성된 통신망에서는 전송하고자 하는 데이터 이외의 추가적인 정보가 각 프로토콜에 의해 덧붙여지거나 분리되게 되어 있으며, 특별히 추가 정보(header 또는 trailer)를 덧붙이는 과정을 캡슐화(encapsulation)[34]

그림 2.1: 인터네트 각 계층 및 프로토콜간의 자료 흐름

라고 한다. 각 프로토콜은 이러한 정보를 바탕으로 해당 프로토콜의 기능을 수행하게 된다. 인터네트 역시 다수의 프로토콜로 구성되어 있으므로 캡슐화 및 분리 과정이 일어나게 된다. 그림 2.2는 사용자 데이터가 TCP, IP, 이더넷을 통해 전송될 때의 캡슐화 과정을 나타낸다. TCP와 IP는 추가 정보로서 적당한 크기의 헤더를 덧붙이며, 이더넷은 헤더(header)와 트레일러(trailer)를 덧붙이게 되어 있다. 반대로 이더넷 프레임(frame)이 사용자에게로 전달될 때에는 이더넷, IP, TCP에 의해 각 추가 정보가 분리된다.

그림 2.2: 인터네트를 통한 데이터 전송 시의 캡슐화

본 장에서는 TCP/IP의 동작을 이해하기 위해 TCP/IP와 이더넷의 개략적 기능과 각 프로토콜의 데이터 캡슐화에 대해 살펴본다. 사용자 인터페이스인 소켓에 대해서도 살펴본다.

제 1 절 이더넷

이더넷(Ethernet)[8]은 DEC(Digital Equipment Corp.), Intel Corp., Xerox Corp.이 개발한 데이터링크 규약(protocol) 및 실제 하드웨어를 말한다. 1982년에 규약이 발표된 후 이더넷은 데이터 링크 계층 규약으로 가장 많이 쓰이는 규약이 되었으며, 기술적으로는 CSMA/CD(Carrier Sense, Multiple Access with Collision Detection)를 바탕으로 10 Mbps 정도의 성능을 나타내고 있다. 그림 2.3은 IP 데이터그램, ARP/RARP 프레임을 이더넷 프레임으로 캡슐화 하는 방법을 나타내고 있다. 6 바이트 크기의 수신/송신 어드레스는 이더넷 하드웨어가 데이터를 원하는 목적지로 보내거나 받기 위한 MAC(Media Access Control) 어드레스이며, type은 캡슐화된 데이터가 IP 데이터그램인지, ARP/RARP 프레임인지를 구별해 주는 숫자이다. 이들 정보 이후에 데이터가 구성되고, 프레임의 제일 끝 부분에는 오류 검사를 위해 CRC(Cyclic Redundancy Check) 바이트가 붙여지게 된다. 예외적으로 ARP/RARP 프레임에는 데이터 바이트 이후에 PAD 바이트가 붙여지게 되는데, 이는 이더넷 데이터의 최소 크기인 46 바이트를 만족시키기 위한 것이다.

그림 2.3: IP 데이터그램과 ARP/RARP 프레임을 이더넷 프레임으로 캡슐화 하기 위한 헤더 및 트레일러 양식

제 2 절 IP

IP[30]는 네트워크 계층에 속하며 데이터그램 전달을 주목적으로 하는 프로토콜이다. IP는 TCP, UDP와 함께 인터네트 상에서 실질적인 데이터 전송을 담당하는 중추적 프로토콜 중의 하나이다. 왜냐하면 응용 계층, TCP, UDP를 비롯하여, 같은 계층에 속하는 ICMP와 IGMP 프로토콜이 모두 IP를 통해 데이터를 전송/수신하기 때문이다. 반면 프로토콜의 서비스 특성은 비연결지향(connectionless)이고, 신뢰적이지 않다(unreliable). 비연결지향이라는 것은 IP가 일련의 패킷에 대한 정보를 모두 유지하지 않고 입력으로 들어오는 데이터그램들을 각각 독립적으로 처리한다는 것이다. 이러한 특성으로 인해 IP에 의해 전달되는 데이터그램들은 순서가 바뀌어 전달될 수도 있게 된다. 예를 들면, 두 개의 패킷을 동일한 수신 호스트로 전달한다고 할 경우에도, IP가 각 데이터그램에 대해 서로 다른 경로를 배정(routing)하고, 이들 패킷이 각 경로를 통해 독자적으로 처리된다고 하면, 수신 측은 송신 순서와 반대로 패킷을 받을 수도 있다는 것이다. 한편, IP에서 제공되지 않는 신뢰성은 TCP와 같은 신뢰성 있는 프로토콜을 통해 보완이 되도록 하고 있다.

2.1 IP 캡슐화

그림 2.4는 트랜스포트 계층의 데이터를 캡슐화 하기 위한 IP 헤더의 구조 및 정보를 나타낸다. 헤더의 크기는 기본적으로 20 바이트이며 옵션의 유무에 따라 크기가 가변적일 수 있다. 이 중 IP의 기능을 이해하는데 중요한 TTL(time-to-live), protocol, checksum, address 필드를 살펴보기로 한다. TTL 필드는 후에 설명할 경로 배정(routing)과 관련이 있는 필드이며, IP 데이터그램이 거칠 수 있는 경로 배정기(router)의 수 또는 네트워크 상에서 IP 데이터그램이 존재할 수 있

는 시간(life-time)을 의미한다. Protocol 필드는 IP를 사용하는 트랜스포토 프로토콜을 지칭하는데 사용된다. 따라서 이 필드는 입력으로 들어온 IP 데이터그램을 적절한 트랜스포트 프로토콜로 역 다중화(demultiplexing) 하는데 필요한 정보이다. Checksum 필드는 IP 헤더의 오류를 검증하는 기능을 하며, 송신(source)/수신(destination) 어드레스(address)는 패킷의 전송 객체와 수신 객체를 지정하는데 쓰인다.

그림 2.4: IP 헤더 양식

2.2 IP 경로 배정

IP의 주된 기능이 데이터그램의 전송/수신(delivery, switching)이므로 데이터그램에 대한 경로 배정(routing)은 IP의 핵심 기능이다. 인터네트의 구조는 통신 주체의 입장에서 볼 때, 통신 주체가 속한 지역 네트워크(local area network)와 외부 네트워크로 나눌 수 있고, 각 네트워크 사이에는 경로 배정기(router)가 존재하게 된다. 그리고 네트워크 상의 호스트 및 경로 배정기는 경로 배정표(routing table)를 바탕으로 데이터그램에 대한 경로 배정을 수행하게 된다. 데이터그램의 전달을 위한 경로 배정은 다음과 같은 3 단계로 이루어진다.

① 수신측 주소가 같은 네트워크 상에 속하는지 라우팅 표를 참조한다. 같은 네트워크에 속하는 것이 확인되면 데이터그램을 곧바로 수신 측으로 보낸다.

② 같은 네트워크에 속하지 않는 주소라면, 경로 배정 표(routing table)에서 수신 측의 네트워크 주소를 검색한다. 일치되는 네트워크 주소를 찾으면 해당 네트워크와 직접적으로 연결된(directly connected) 경로 배정기로 데이터그램을 전달한다.

③ ①, ②의 과정에 실패할 경우 기본 경로 배정기(default router, 어떤 경로 배정기가 기본 경로 배정기인지는 미리 정해져 있다.)로 데이터그램을 보내게 되며, 이 단계에서도 경로 배정이 실패할 경우 해당 데이터그램은 버려지게 된다.

이와 같은 경로 배정 방식을 'hop-by-hop' 방식이라 하며, 살펴본 바와 같이 각 호스트 및 경로 배정기는 인접한 호스트 및 경로 배정기에 대한 정보만을 바탕으로 경로 배정을 수행하게 된다

제 3 절 TCP

TCP[31]는 트랜스포트 계층의 프로토콜로 연결지향(connection-oriented), 바이트 열(byte stream) 방식의 전송이며, 신뢰성 있는(reliable) 프로토콜이다. 연결지향이란 것은 TCP를 이용하여 통신하는 두 개의 응용 프로그램은 데이터 전송을 하기 전에 TCP 연결을 설정해야 함을 말한다. 신뢰성 있는 프로토콜이란 것은 TCP가 사용자 데이터를 오류가 없도록 전송해 준다는 의미이다. 바이트 열 전송이라는 뜻은 데이터의 전송을 고정 길이 레코드(fixed size record) 구조가 아닌 가변 길이 레코드 구조를 통해 수행한다는 것이다. 신뢰성 있는 데이터 전송은 다음

과 같은 방법을 통해 이루어지도록 하고 있다.

- 자료의 조각화 (segmentation)
- 수신된 자료에 대한 acknowledgment(이하 ACK 이라 함) 세그먼트의 전송
- 재전송(retransmission)을 위한 타이머의 유지
- 체크섬 계산
- 순서 정렬 (ordering)
- 중복 세그먼트 검출(detection of duplicate segment)
- 흐름 제어(flow control)의 제공

자료의 조각화는 사용자가 전송하려는 데이터를 네트워크의 상태를 고려하여 적절한 크기로 분할하여 보내는 것이다. 이 때의 데이터 조각을 세그먼트라고 한다. ACK 세그먼트의 전송은 수신 측의 데이터 수신 여부를 송신 측에게 알리는 기능을 한다. 따라서 송신 측은 ACK 세그먼트의 수신을 통해 데이터 전송의 성공 여부를 알 수 있다. 재전송을 위한 타이머는 송신측이 ACK 세그먼트를 무한히 기다리지 않게 하도록 하는 기법이다. 타이머에 의해 일정한 시간이 경과되면 송신 측은 데이터 전송에 오류가 있다고 간주를 하고 이미 보낸 세그먼트를 다시 보내게 된다. 체크섬 계산은 수신자가 수신한 세그먼트의 오류 검출하기 위한 방법이다. 체크섬 계산 결과 오류가 발생한 세그먼트는 데이터로서 받아들이지 않게 된다. 이는 잠재적으로 세그먼트의 재전송을 유발하게 된다. 순서 정렬은 수신되는 TCP 세그먼트들을 원래의 순서대로 배열을 하는 것이다. 재배열이 필요한 이유는 TCP 세그먼트가 순서배열을 고려하지 않는 IP 데이터그램의 형태로 전달되기

때문이다. 그리고 중복 수신된 세그먼트를 검출한 후 버리는 기능을 한다. 마지막으로 TCP는 원활한 데이터 흐름을 위한 흐름 제어 기술을 제공하고 있다.

3.1 TCP 캡슐화

TCP 헤더의 구성 양식은 그림 2.5과 같다. TCP 헤더의 크기는 기본적으로 20 바이트이며 옵션의 존재 여부에 따라 추가적인 바이트가 필요할 수도 있다. 헤더의 정보 중 TCP/IP의 데이터 전송, 수신을 위해 중요한 정보는 port number, sequence number, ACK number, checksum 등이다. Port number는 송/수신 측의 응용 프로그램을 서로 구분하기 위해 사용하는 번호이다. Sequence number는 전송하고자 하는 세그먼트의 첫 번째 바이트가 전체 데이터 스트림(stream) 중 몇 번째에 해당되는 지를 나타낸다. ACK number는 세트먼트의 수신 측에 의해 설정되어 돌아오는 값이며, 송신측의 다음 번 sequence number를 나타내는 값이다. Checksum은 헤더를 포함한 전체 TCP 세그먼트에 대한 오류 검증을 위해 사용된다.

그림 2.5: TCP 헤더 양식

3.2 TCP 데이터 교환

전체적인 TCP의 동작을 이해하기 위해서는 TCP의 상태 전이 및 연결의 설정/해제, 자료 흐름과 흐름 제어[1], 타이머, congestion control[11] 등의 복잡한 과정을 이해해야 한다. 그러나 본 절에서는 본 논문의 실험 환경을 이해하기 위한 목적으로 데이터 세그먼트의 전송 방법[6]에 대해서만 살펴본다. 그림 2.6은 하나의 세그먼트를 송신측으로부터 수신 측으로 전달하는 과정을 나타내고 있다. 송신측이 세그먼트를 전송하고 나면 송신측은 수신 측으로부터의 ACK 세그먼트를 기다리게 되고, 수신 측은 전달받은 세그먼트에 대한 ACK 세그먼트를 송신측에 전송하게 되다. 이 과정에 필요한 정보 중 중요한 것은 각 세그먼트의 헤더에 포함된 sequence number, ACK number, checksum이다. 수신 측은 세그먼트를 전달받으면, checksum을 검사하여 세그먼트에 오류가 있는가를 검사하게 된다. 오류가 있다면 세그먼트를 무시하고 버리게 된다. 세그먼트에 오류가 없으면 sequence number를 검사하여 세그먼트의 중복성을 검사한다. 세그먼트가 중복되어 들어왔을 경우에는 세그먼트를 버리게 된다. 다음으로는 순서 검사를 하게 된다. 검사 결과 순서에 맞지 않게 전달된 세그먼트는 순서정렬을 위해 버퍼에 저장된다. 순서정렬 방법은 구현 알고리즘에 따라 조금씩 다르므로 여기서는 언급하지 않겠다. 순서 검사에 의해 올바른 순서로 세그먼트가 전달되었다면, 수신 측은 전달받은 세그먼트의 sequence number와 수신된 데이터 크기를 계산하여, 다음 번에 수신할 세그먼트의 sequence number를 계산해 낸다. 계산된 값은 ACK 세그먼트의 ACK number 필드에 저장되어 송신측에 전달되게 된다. 송신측은 수신 측으로부터 받은 ACK number를 기준으로 다음 번 전송할 세그먼트의 sequence number 및 세그먼트를 생성하게 된다. 다음 번 세그먼트가 생성되면, 전송 및 수신 측은 전술한

방법과 동일한 방법으로 데이터 전송을 계속 수행하게 된다.

제 4 절 소켓

지금까지 설명한 TCP/IP, 이더넷은 일반적으로 운영체제 내의 커널(kernel)에 구현되기 때문에 응용 프로그램은 프로토콜에 접근하기 위한 인터페이스(API:Application Program Interface)를 필요로 하게 된다. 이에 각 운영체제는 프로토콜에 접근을 하기 위한 인터페이스를 개발하여 왔고, 대표적인 것이 버클리 소켓(Berkeley socket)[36]과 TLI(Transport Layer Interface)이다. 본 논문에서는 이 중 소켓을 채택하였다.

소켓의 원론적인 기능은 앞서 설명한 바와 같이 응용 프로그램과 커널 사이에 위치하여 프로토콜 서비스를 중계하는 것이며, 기술적으로 중요한 것은 응용 프로그램에 일관된 인터페이스를 제공하는 것이다. 커널은 다수의 프로토콜로 구성되어 있기 때문에 프로토콜에 접근하기 위한 방법은 각 프로토콜마다 다르게 된다. 그러나 소켓을 이용하면 거의 동일한 양식을 통해 각각의 프로토콜에 접근할 수 있게 되며, 이러한 양상은 개개의 운영체제 및 하드웨어에 독립적이다. 데이터의 전송 및 수신도 운영체제에서 제공하는 논리적인 입/출력 기능을 통해 수행할 수 있다. 이같은 소켓의 일관성은 네트워크 응용 프로그램의 이식성을 높였고, 많은 응용 프로그램들이 TCP/IP 및 타 프로토콜을 이용할 수 있도록 하였다.

소켓과 커널과의 접속을 개념적으로 나타낸 것이 그림 2.7이다. 소켓 인터페이스(socket system call interface)는 운영체제의 사용자 영역에 위치하며, 응용 프로그램이 커널에 접근할 수 있도록 해주는 함수들이다. 소켓 구현(socket implementation)은 커널 영역에 위치한 함수들로 트랜스포트 프로토콜로의 접속 및 데이터 전송을 수행하게 된다. 데이터의 전송 시에는 소켓 구현이 전송에 이용할 프로토

그림 2.6: TCP에서의 세그먼트 전송

그림 2.7: 소켓과 커널의 접속 및 데이터 흐름

콜을 선택하고, 사용자 영역에 위치한 데이터를 커널 내의 소켓 버퍼로 복사하게 된다. 데이터가 복사된 후, 소켓은 해당 트랜스포트 프로토콜에 데이터 전송을 요청하게(user request) 되고, 이 요청에 의해 실질적인 데이터 전송이 일어나게 된다. 수신은 트랜스포트 프로토콜이 네트워크 계층으로부터 데이터를 받아 소켓 버퍼에 저장한 후, 데이터 도착을 알리는 정보(signal)를 소켓에 보내는 것으로부터 시작된다. 송신 측이 전송한 데이터는 이미 소켓 버퍼에 저장된 상태이므로, 소켓은 이 데이터를 응용 프로그램으로 복사하면 된다.

제 3 장

성능 분석 방법 및 도구

제 1 절 성능 분석 방법 비교

통신 프로토콜의 성능 분석 방법으로는 수학적 모델을 이용하는 방법[23, 22], 소프트웨어적인 분석 도구를 사용하는 방법[9], 독립적으로 동작하는 하드웨어 도구를 이용하는 방법[27] 등이 있다. 이들 방법은 성능 측정을 위한 비용, 구현의 용이성, 얻을 수 있는 정보의 양, 정확도 등의 측면에서 장·단점을 가지게 된다. 일반적으로, 소프트웨어적인 구현은 비용 측면에서는 유리하나 정확도가 떨어지고, 하드웨어적인 구현은 비용이 많이 드나 정확도가 우수한 특징이 있다. 먼저 수학적 모델[23]을 이용하는 방법은, 먼저 평가 대상으로부터 성능에 영향을 미치게 될 부분(module)을 추출해 내고(feature abstraction) 이에 대한 수학적 모델(formal specification)을 수립한다. 다음으로 성능 평가 기준을 수립한 후 이에 대한 세분화된 모델링을 수행한다(specification enhancement). 모듈에 대한 모델이 수립되면 실험 환경에 구현을 하게 된다. 구현된 실험 환경 하에서 환경 변수를 달리하여 시뮬레이션을 수행함으로써 성능 측정을 하게 된다. 수행 결과로부터 시스템에 대한 최적의 환경 변수를 찾아내거나, 성능이 좋지 않은 모듈을 찾아내게 된다. 성능이 좋지 않은 모듈에 대해서는 새로운 설계, 즉 새로운 모델링을 하게 되고, 앞서

와 같은 시뮬레이션을 반복하게 된다. 이와 같은 방법은 모듈간의 상대적인 성능 비교와 평균적인 성능을 알 수 있으나 실제 구현에 관한 요소를 정확히 반영할 수 없으므로 시스템의 절대적인 성능 평가에는 어려움이 있다고 볼 수 있다. 즉, 모듈 내의 병목 부분을 정확히 밝혀 내기 힘든 단점이 있다. 한편, 이와 같은 이론적인 성능 평가의 한계를 극복하기 위해 실측을 하려는 시도가 계속 되어 왔다[9, 27]. 실측의 장점은 각 모듈에 대한 실제 수행시간을 알 수 있고, 좀더 세밀한 측정을 수행하기 때문에 병목 부분에 대해 자세한 분석이 가능하다는 것이다. 따라서 수행 시간을 실시간에 측정하기 위한 소프트웨어[9] 또는 하드웨어[27]가 제작되었다.

소프트웨어 측정 도구를 이용하는 방법[9]은 운영체제 내에 시간을 측정할 수 있는 디바이스 드라이버를 작성하는 것이다. 시간 측정은 운영체제 내의 클럭(clock)을 사용하게 된다. 소프트웨어 측정 도구를 이용하는 방법의 장점은 개발의 용이성이라 할 수 있다. 운영체제의 커널 내에 측정 도구 즉, 디바이스 드라이버를 작성하고 측정하고자 하는 커널 또는 응용 프로그램의 모듈 안에 시간 측정을 시작/중지하기 위한 코드를 삽입하는 것이다. 따라서 세부적인 측정이 가능하도록 해준다. 그러나 측정을 수행하는 동안 컴퓨터의 운용 환경에 따라 부정확한 자료를 얻을 수도 있다. 예를 들면, 측정 모듈의 수행과 직접적인 관련이 없는 인터럽트 처리 시간이 실측 자료에 반영될 수 있다. 그리고 운영체제가 제공하는 클럭의 측정 단위가 정밀하지 않아서 – 예를 들면 4.4BSD UNIX의 경우 7.8 ms 정도의 정밀도를 갖는 클럭을 사용함[26] – 고속으로 동작하는 모듈의 속도를 정확히 측정하기 어려운 단점이 있다.

이러한 소프트웨어적인 측정의 단점을 보완하기 위해 하드웨어를 이용하는 방법이 등장하게 되었다[27]. 하드웨어 제작에 많은 비용이 소요되고, 이식성(portability)

및 융통성(flexibility)이 없는 것이 단점이지만, 소프트웨어를 이용하는 방법에 비해 정확한 측정을 할 수 있는 것이 장점이다. 이와 같은 방법은 우선 정밀한 시간 측정이 가능한 하드웨어 타이머(timer)를 제작하고, 이를 컴퓨터에 장착 가능한 보드(board)로 만드는 것이다. 그리고 이 하드웨어를 구동하기 위한 소프트웨어를 작성하고 측정하고자 하는 프로그램의 모듈 내에 측정을 시작/중지할 수 있는 코드를 삽입하는 것이다. 소프트웨어적인 측정과 달리 외부 환경의 영향을 배제할 수 있으므로 보다 정확한 측정을 할 수 있다. 100 ns의 정밀도를 갖는 타이머 보드를 이용하여 통신프로토콜의 수행 성능을 분석한 사례가 있다[27].

제 2 절 성능 분석 도구

본 논문에서 사용한 성능 측정 및 분석 도구는 펜티엄 프로세서에 내장된 카운터(이하 '펜티엄 카운터' 또는 '카운터'라 하겠다.)들이다. 지금까지 설명한 성능 측정 방법들은 구현의 용이성, 비용, 이식성, 융통성, 정확도를 동시에 만족할 수 없었다. 그러나 펜티엄 카운터는 이들 조건을 모두 만족하며, 특히 정확도에 있어서는 기존의 방법에 비해 월등히 우수하다. 그리고 기존의 방법들이 시스템의 성능을 시간 측면에서만 측정할 수 있었던 것에 반해 펜티엄 카운터는 프로그램의 복잡도 및 메모리 대역폭에 대해서도 측정을 할 수 있다.

2.1 펜티엄 내부 카운터

인텔 펜티엄 프로세서는 기존의 8086, 80286, 80386, 80486 프로세서와 호환성을 가지며, 펜티엄만의 독자적인 코드를 수행할 수 있도록 설계된 프로세서이다. 본 논문에서 사용한 펜티엄 카운터 인스트럭션은 펜티엄 프로세서만의 인스트럭션이며, 프로세서 내의 레지스터 집합인 TSC(Time-Stamp Counter), MSRs(Model[Machine]-

Specific Registers)의 사용법과 연관되어 있다. 이들은 프로세서의 상태를 저장하고 있는 레지스터 집합이며, 제품 개발자 및 사용자는 이를 이용해 프로세서의 시험(testability), 실행 추적(execution tracing), 성능 측정(performance monitoring), 프로세서의 오류 확인(machine check errors)을 할 수 있다. 레지스터의 크기는 64비트이며 펜티엄 프로세서는 이 레지스터에 데이터를 쓰거나 읽기 위한 인스트럭션인 RDTSC(Read TSC), RDMSR(Read MSR), WRMSR(Write MSR)을 제공하고 있다.

인텔사는 펜티엄 프로세서를 제품으로 발표한 후 일반인에게는 이들 레지스터에 대한 기능과 사용법을 공개하지 않았다. 다만 인텔과 비공개 원칙에 합의한 몇몇 프로그래머들에게만 공개를 하였고, 이들에 의해 펜티엄 카운터를 구동하는 프로그램들이 제작되었다. 그러나 참고문헌 [32]에 의해 펜티엄 카운터의 존재가 알려지기 시작하였고, 참고문헌 [24]는 기계어로 된 카운터 구동 프로그램을 분석하여 카운터의 사용법을 밝혀 내고 상술하였다. 이로부터 펜티엄 카운터는 성능평가의 중요한 수단으로 쓰이게 되었다. 본 논문에서는 참고문헌 [32, 24]에 의해 알려진 한 개의 TSC(이하 싸이클 카운터라 함)와 두개의 MSR(이하 이벤트 카운터라 함)을 사용하였다.

2.1.1 펜티엄 싸이클 카운터

펜티엄 프로세서의 싸이클 카운터인 타임 스탬프 카운터(Time Stamp Counter)는 2.1.2절에서 설명할 MSR(Model[Machine]-Specific Register)의 일종으로 머리 글자가 의미하는 것처럼 시스템, 즉 프로세서의 수행 시간을 저장하고 있는 레지스터이고 참고문헌 [32]에 의해 알려지게 되었다. 여기서 말하는 프로세서의 수행 시간이란 프로세서의 클록 틱(clock tick) 수를 말하며 펜티엄 프로세서는 각 클록

틱마다 이 레지스터의 값을 증가시키게 되어있다. 따라서 이 레지스터는 프로세서의 클록 속도로 동작하게 되며 정밀도 역시 프로세서의 클록과 일치하게 된다. 정밀도와 더불어 이 레지스터가 중요한 이유는 동작 특성이다. TSC는 클록과 동기화(synchronization) 되어 동작하므로 프로세서가 인스트럭션을 수행하지 않는 동안에도 지속적으로 증가된다. 따라서 인스트럭션의 수행시간 및 프로세서의 휴지(idle) 상태를 포함한 실제 수행 시간을 산출할 수 있는 것이다.

TSC에 대해서는 RDTSC(Read TSC) 인스트럭션이 존재하며, 그림 3.1은 TSC 값을 읽어 내는 어셈블리 코드의 예이다. RDTSC에 대한 니모닉은 일반 어셈블러에서는 제공하지 않으므로 1행부터 3행을 통해 사용자가 나름대로의 수행 매크로를 정하게 된다. 즉, 2행의 코드는 RDTSC의 기계어 코드 값을 나타낸다. 6행의 RDTSC 인스트럭션이 수행되면 현재의 TSC 값이 32 비트씩 나뉘어 각각 EDX, EAX 레지스터로 전달이 되게 된다. 사용자는 7, 8행과 같이 이 값을 적절한 위치로 복사하면 된다. TSC에 값을 설정하는 방법은 MSR을 기술하면서 하기로 한다.

```
1:      RDTSC MACRO
2:          db 0fh, 031h
3:      ENDM
4:      ; End of macro definition
5:
6:      RDTSC       ; execute RDTSC instruction
7:      mov hi, edx ; save higher order 32 bits
8:      mov lo, eax ; save low order 32 bits
```

그림 3.1: TSC 값을 읽기 위한 어셈블리 코드의 예

2.1.2 펜티엄 이벤트 카운터

펜티엄 프로세서의 이벤트 카운터는 프로세서 내의 MSR(Machine[Model] Specific Register)이며, 펜티엄 프로세서 사용자 지침서[10]는 다수의 MSR 중 번호 0,

1에 해당되는 MCAR(Machine Check Address Register), MCTR(Machine Check Type Register)에 대해[2, 10] 언급하고 나머지의 MSR들에 대해서는 비공개(non-disclosure), 부정한 사용(illegal use), 또는 예약됨(reserved)이라 하여 일반인에게는 공개하지 않았다. 그러나 참고문헌 [24]는 알려지지 않은 MSR 중 카운터로 쓰이는 3 개의 MSR에 대한 기능과 사용법을 밝혀 내었다.

MSR로의 접근은 RDMSR/WRMSR 인스트럭션을 통해 이루어진다. MSR의 값을 읽기 위해서는 접근하고자 하는 MSR의 번호를 범용 레지스터인 ECX에 넣고, RDMSR 인스트럭션을 수행하면 된다. 수행 결과로 해당 MSR의 값이 범용 레지스터인 EDX, EAX에 저장되게 된다. 반대로 사용자가 MSR의 값을 갱신하려면 EDX, EAX에 갱신하고자 하는 값을 넣고, ECX에 원하는 MSR 번호를 넣은 후 WRMSR 인스트럭션을 수행하면 된다.

```
 1:       RDMSR MACRO
 2:              db 0fh, 032h
 3:       ENDM
 4:
 5:       WRMSR MACRO
 6:              db 0fh, 030h
 7:       END
 8:       ; End of macro definition
 9:
10:       mov ecx, 12h ; select MSR to read from
11:       RDMSR        ; execute RDMSR instruction
12:       mov hi, edx  ; save higher order 32 bits
13:       mov lo, eax  ; save low order 32 bits
14:
15:       mov ecx, 11h ; select MSR to write to
16:       mov edx, hi  ; prepare higher order 32 bits
17:       mov eax, lo  ; prepare low order 32 bits
18:       WRMSR        ; execute WRMSR instruction
```

그림 3.2: MSR에 접근하기 위한 어셈블리 코드의 예

수행 예를 그림 3.2에 보였다. 1–8행은 그림 3.1에서와 같이 RDMSR, WRMSR

의 기계어 코드를 나타낸다. 10행은 값을 읽어내고자 하는 MSR의 인덱스 번호를 설정하는 코드이며, 11행의 RDMSR 인스트럭션에 의해 64 비트의 MSR 값이 EDX, EAX에 저장되게 된다. 마찬가지로 15–18행의 코드는 갱신하고자 하는 MSR을 선택하고(15행), 써 넣고자 하는 데이터를 EAX, EDX에 설정(16, 17행)한 후, 마지막으로 WRMSR 인스트럭션을 수행하는 것을 보여주고 있다.

참고문헌 [32, 24]에 의해 밝혀진 MSR은 인덱스가 16, 17, 18, 19인 레지스터이다(16진수로 10h, 11h, 12h, 13h). 네 개의 MSR 중 16번은 TSC이고(그러므로 TSC에 값을 설정하기 위해서는 WRMSR 인스트럭션을 이용하면 된다.) 17번은 MSR 18, 19 번을 제어하기 위한 레지스터이다. 18, 19번 MSR은 동일한 기능을 하는 두개의 MSR로, 각각은 TSC와 달리 프로세서 내에서 발생한 이벤트의 수를 – 예를 들면 메모리 참조 수, 인스트럭션 수, 인터럽트 수, 캐쉬 미스 수 등의 이벤트 수 – 저장하게 된다. 이들 MSR을 이용해 측정할 수 있는 이벤트는 38 가지가 알려져 있으며[24], 표 3.1에 구체적으로 나타내었다.

MSR을 이용해 이벤트를 측정하는 것은 크게 두 가지 단계로 나누어 볼 수 있다. 첫 번째 단계는 MSR 17번과 WRMSR 인스트럭션을 이용해 측정하고자 하는 이벤트(싸이클)를 설정하는 단계이고, 두 번째 단계는 RDMSR 인스트럭션을 이용해 MSR 18, 19번의 값을 읽어내는 것이다. 이벤트를 설정하기 위해 MSR 17번에 전달되어야 하는 데이터 형식은 그림 3.3과 같다. 64 비트 중 하위 32 비트만을 사용하며 32 비트는 16 비트씩 나뉘어 MSR 18, 19에 대한 이벤트를 각각 설정할 수 있게 된다. 16 비트의 값 중 하위 여섯 개 비트는 측정하고자 하는 이벤트의 종류를 설정하기 위해 사용된다. 각 이벤트의 번호는 표3.1의 인덱스 번호를 뜻한다. 7 번째 비트는 운영체제의 링 0–2(ring 0–2, kernel mode)에서 일어난 이벤트를 측정하려는 목적으로 설정하고 8 번째 비트는 운영체제의 링 3(ring 3, user mode)에

표 3.1: 펜티엄 카운터를 이용해 측정 가능한 이벤트들

Index(Hex.)	Event Name
0	Data read
1	Data write
2	Data TLB (translation look-aside buffer) miss
3	Data read miss
4	Data write miss
5	Write (hit) to M or E state lines
6	Data cache lines written back
7	Data cache snoops
8	Data cache snoop hits
9	Memory accesses in both pipes
A	Bank conflicts
B	Misaligned data memory references
C	Code read
D	Code TLB miss
E	Code cache miss
F	Any segment register load
12	Branches
13	BTB (branch target buffer) hits
14	Taken branch or BTB hit
15	Pipeline flushes
16	Instructions executed
17	Instructions executed in the v-pipe
18	Bus utilization (clocks)
19	Pipeline stalled by write backup
1A	Pipeline stalled by data memory read
1B	Pipeline stalled by write to E or M line
1C	Locked bus cycle
1D	I/O read or write cycle
1E	Noncacheable memory references
1F	AGI (Address Generation Interlock)
22	Floating-point operations
23	Breakpoint 0 match
24	Breakpoint 1 match
25	Breakpoint 2 match
26	Breakpoint 3 match
27	Hardware interrupts
28	Data read or data write
29	Data read miss or data write miss

서 일어난 이벤트를 측정하기 위해 설정한다. 9 번째 비트는 해당 이벤트 카운터가 MSR에 이벤트의 수를 반환할 것인지 싸이클 수를 반환할 것인지를 결정한다. 이벤트의 수는 지금까지 언급했던 이벤트를 의미하나, 싸이클 수는 TSC에서 살펴보았던 싸이클의 수와 조금 다른 의미를 지닌다. 여기서 말하는 싸이클 수는 해당 이벤트가 사용한 싸이클의 수이다. 즉, 특정 이벤트를 수행하기 위해 사용된 순수한 싸이클의 수를 말하며, 또 다른 의미로는 해당 이벤트만을 수행하는데 소요된 시간이라고 할 수 있다.

그림 3.3: MSR 18, 19를 제어하기 위한 MSR 17의 자료 형식

2.2 커널 내부의 성능을 측정하기 위한 펜티엄 드라이버

앞서 예를 든 카운터 구동 코드는 카운터의 동작을 이해하기 위해 간단히 구성한 코드이다. 따라서 실제 응용에 사용하기 위해서는 일관된 사용자 인터페이스를 제공하는 디바이스 드라이버가 필요하다. FreeBSD 상에서 동작하는 디바이스 드라이버가 제작이 되었으나 [4], 이들 드라이버는 응용 프로그램의 성능 평가를 주목적으로 하였기 때문에 본 논문에서 사용하기는 부적절하였다. 이에 본 논문에서는 커널 내부의 성능을 측정할 수 있는 드라이버를 작성하여 실험에 이용

하였다.

2.2.1 카운터 값을 저장하기 위한 자료 구조

그림 3.4는 성능 측정 결과를 저장하기 위한 자료 구조로, TSC와 MSR을 저장하기 위한 변수와 성능 측정의 동기화를 위한 변수, 측정 시의 환경을 저장하기 위한 변수를 제공하고 있다. 본 논문에서 수행하는 성능 측정은 측정 시간이 짧으므로 TSC, MSR의 64 비트 중 하위 32 비트만을 이용하였다. 그림 3.5는 카운터 값을 저장하기 위한 메모리 영역을 할당하는 부분이다. 최대 10(MAX_POOL) 개의 저장 장소를 설정하였고, 전체 커널 영역에 대해 전역 변수로 정의하여 커널 내의 어떠한 부분에서도 참조가 가능하도록 하였다.

```
 1:    #define MAX_POOL        10
 2:
 3:    #define CALL_NO         207
 4:
 5:    #define MAX_CMD         4
 6:    #define DUMP            0
 7:    #define CLEAR           1
 8:    #define SET_EVENT       2
 9:    #define GET_EVENT       3
10:
11:    #define MAX_STATUS      4
12:    #define FAIL            0
13:    #define INIT            1
14:    #define RUNNING         2
15:    #define SET             3
16:
17:
18:    struct  cnt_pool {
19:            unsigned int tsc;
20:            unsigned int msr0;
21:            unsigned int msr1;
22:            int     para;
23:            int     status;
24:    };
```

그림 3.4: 카운터 값을 저장하기 위한 자료 구조

```
 1: #ifdef  PCNT
 2: #include <machine/pcnt_var.h>
 3: struct  cnt_pool pool[MAX_POOL];
 4: #endif
 5:
 6: /*
 7:  * System startup; initialize the world, create process 0, mount root
 8:  * filesystem, and fork to create init and pagedaemon.  Most of the
 9:  * hard work is done in the lower-level initialization routines including
10:  * startup(), which does memory initialization and autoconfiguration.
11:  */
12: void
13: main(framep)
14:         void *framep;
15: {
16:         register struct proc *p;
17:         register struct filedesc0 *fdp;
18:         register int i;
19:         int s, rval[2];
20:
21:   /*
22:   * Initialize the current process pointer (curproc) before
23:    ......
24:   /* omitted remainder of main program */
```

그림 3.5: 전역 변수로 정의된 펜티엄 카운터의 저장 영역

2.2.2 성능 측정을 수행하기 위한 함수

그림 3.6은 성능 측정을 시작하기 위해 카운터의 값을 '0'으로 초기화 하는 함수이다. 성능 측정을 수행하는 동안 인터럽트의 발생을 막기 위해 7행과 26행의 어셈블리 코드를 사용하였다. 9–18행은 성능 측정을 동기화 하거나 사용자의 성능 측정 오류를 검출하기 위한 코드이다. 동기화가 올바르게 수행되고 사용자의 측정 오류가 없을 경우 19행에서 성능 측정과 관련된 추가적인 정보를 저장한 후, 22–24행의 WRMSR 인스트럭션에 의해 TSC, MSR을 '0'으로 초기화 한다.

그림 3.7은 카운터 값을 읽어내는 함수이다. 먼저 11–13행을 통해 카운터 값을 읽은 후 성능 측정 시작 시와 마찬가지로 인터럽트에 대한 처리(9행, 32행)를 수행하고 동기화 및 오류 검사(15–26행)를 수행한다. 오류가 없을 경우 변수 `'id'`로 지정된 저장 장소에 카운터 값을 저장하게 된다(28–30행).

```
 1:    inline  void
 2:    start_cnt(id, para)
 3:            int id;
 4:            int para;
 5:    {
 6:            /* Disable interrupts between clears */
 7:            __asm __volatile ("cli" ::);
 8:
 9:            if (pool[id].status != INIT ){
10:                    if ( pool[id].status != SET )
11:                            pool[id].status = FAIL;
12:                    /* aleady set */
13:
14:                    __asm __volatile ("sti" ::);
15:                    return;
16:            }
17:
18:            pool[id].para = para;
19:            pool[id].status = RUNNING;
20:
21:
22:            wrmsr(0x10, 0, 0);
23:            wrmsr(0x12, 0, 0);
24:            wrmsr(0x13, 0, 0);
25:
26:            __asm __volatile ("sti" ::);
27:
28:    }
```

그림 3.6: 카운터 값을 초기화 하는 함수

```
 1:    inline  void
 2:    stop_cnt(id)
 3:            int id;
 4:    {
 5:
 6:            unsigned int tsc, msr0, msr1, dummy;
 7:
 8:            /* Disable interrupts between clears */
 9:            __asm __volatile ("cli" ::);
10:
11:            rdtsc(&dummy, &tsc);
12:            rdmsr(0x12, &dummy, &msr0);
13:            rdmsr(0x13, &dummy, &msr1);
14:
15:            if ( pool[id].status != RUNNING ){
16:
17:                    if ( pool[id].status != SET )
18:                            pool[id].status = FAIL;
19:                    /* already set */
20:
21:                    __asm __volatile ("sti" ::);
22:                    return ;
23:
24:            }
25:
26:            pool[id].status = SET;
27:
28:            pool[id].tsc   = tsc;
29:            pool[id].msr0  = msr0;
30:            pool[id].msr1  = msr1;
31:
32:            __asm __volatile ("sti" ::);
33:
34:    }
```

그림 3.7: 카운터 값을 읽어내는 함수

2.2.3 응용 프로그램과의 접속

커널의 성능 측정 결과를 응용 프로그램으로 전달하거나 카운터를 제어하기 위한 방법으로는 시스템콜(system call)[36]을 사용하였다. 시스템콜은 커널이 응용 프로그램에 제공하는 일종의 함수로, 각 시스템콜에는 고유한 시스템콜 번호가 부여되어 있다. FreeBSD UNIX는 그림 3.8에서 보는 바와 같이 모두 220 개의 시스템콜을 제공하고 있다. 본 논문에서는 서비스 함수가 지정되어 있지 않은 시스템콜 번호 207을 이용해 데이터의 전달 및 카운터에 대한 제어를 수행하도록 하였다. 18행의 'pcnt_control'이 본 논문에서 작성한 시스템콜의 함수 이름이며, 숫자 '3'은 이 시스템콜로 전달될 파라미터의 수이다.

그림 3.9는 본 논문에서 작성한 시스템콜의 실제 코드이다. 이벤트 카운터의 측정 이벤트를 설정(SET_EVENT)하거나 현재의 설정 상태를 알아내는(GET_EVENT) 코드와 함께, 카운터 값을 저장하기 위한 메모리 영역을 초기화(CLEAR) 하고, 측정한 데이터를 응용 프로그램으로 전달하는(DUMP) 코드를 포함하고 있다. 8–10행은 응용 프로그램으로부터 시스템콜로 전달될 파라미터 목록이다. 'cmd'는 시스템콜을 통해 수행할 명령(SET_EVENT/GET_EVENT/CLEAR/DUMP)을 나타내며, 'ptr'은 카운터 값을 저장하게 될 배열(응용 프로그램의)에 대한 포인터이다. 'flag'는 이벤트 카운터의 측정 이벤트를 설정하거나 정보를 얻어내기 위한 변수이다. 24–48행이 나타내는 것처럼 시스템콜은 'cmd'로 전달되는 명령의 종류에 따라 동작하도록 되어 있다. 명령어가 측정이벤트를 바꾸는 것일 경우 26행의 'wrmsr'을 통해 측정 이벤트를 변경하게 된다. 반대로, 현재의 설정 상태를 얻어내기 위해서는 'rdmsr'을 통해 정보를 얻어내게 된다. 32–33행은 저장 영역을 초기화 하는 부분이며, 36–43행은 측정된 값을 응용 프로그램으로 전달하는 부분이다.

```
 1: struct sysent sysent[] = {
 2:         { 0, nosys },                   /* 0 = syscall */
 3:         { 1, exit },                    /* 1 = exit */
 4:         { 0, fork },                    /* 2 = fork */
 5:         { 3, read },                    /* 3 = read */
 6:         { 3, write },                   /* 4 = write */
 7:         { 3, open },                    /* 5 = open */
 8:         { 1, close },                   /* 6 = close */
 9:         { 4, wait4 },                   /* 7 = wait4 */
10:         { compat(2,creat) },            /* 8 = old creat */
11:         { 2, link },                    /* 9 = link */
12:         { 1, unlink },                  /* 10 = unlink */
13:
14:  /* omitted entris from 11 to 205 intentionally */
15:
16:         { 0, nosys },                   /* 206 = nosys */
17: #ifdef PCNT
18:         { 3, pcnt_control},            /* 207 = pcnt control routine */
19: #else
20:         { 0, nosys },                   /* 207 = nosys */
21: #endif
22:         { 0, nosys },                   /* 208 = nosys */
23:         { 0, nosys },                   /* 209 = nosys */
24:         { 0, lkmnosys },                        /* 210 = lkmnosys */
25:         { 0, lkmnosys },                        /* 211 = lkmnosys */
26:         { 0, lkmnosys },                        /* 212 = lkmnosys */
27:         { 0, lkmnosys },                        /* 213 = lkmnosys */
28:         { 0, lkmnosys },                        /* 214 = lkmnosys */
29:         { 0, lkmnosys },                        /* 215 = lkmnosys */
30:         { 0, lkmnosys },                        /* 216 = lkmnosys */
31:         { 0, lkmnosys },                        /* 217 = lkmnosys */
32:         { 0, lkmnosys },                        /* 218 = lkmnosys */
33:         { 0, lkmnosys },                        /* 219 = lkmnosys */
34: };
```

그림 3.8: UNIX가 유지하는 시스템콜 정보

```
 1:    /* 207th system call */
 2:    /* by hkim */
 3:    #ifdef PCNT
 4:    #include <machine/pcnt_var.h>
 5:    #include <machine/pcnt_proto.h>
 6:
 7:    struct pcnt_args {
 8:            int     cmd;
 9:            struct  cnt_pool *ptr;
10:            unsigned *flag;
11:    };
12:
13:    int
14:    pcnt_control(p, args, ret_val)
15:            struct  proc *p;
16:            struct  pcnt_args       *args;
17:            int     *ret_val;
18:    {
19:            int i;
20:            unsigned int dummy;
21:
22:            __asm __volatile ("cli" ::);
23:            switch (args->cmd){
24:            case    SET_EVENT:
25:                    wrmsr(0x11, 0, *(args->flag));
26:                    break;
27:            case    GET_EVENT:
28:                    rdmsr(0x11, &dummy, args->flag);
29:                    break;
30:            case    CLEAR:
31:                    for ( i = 0 ; i < MAX_POOL ; i ++){
32:                    pool[i].status = INIT;
33:                    pool[i].tsc = pool[i].msr0 = \
34:                    pool[i].msr1= pool[i].para = 0;}
35:                    break;
36:            case    DUMP:
37:                    for ( i = 0 ; i < MAX_POOL ; i ++)
38:                    {
39:                            args->ptr[i].tsc    = pool[i].tsc;
40:                            args->ptr[i].msr0   = pool[i].msr0;
41:                            args->ptr[i].msr1   = pool[i].msr1;
42:                            args->ptr[i].status = pool[i].status;
43:                            args->ptr[i].para   = pool[i].para;
44:                    }
45:                    *ret_val = 0 ;
46:                    break;
47:            default:
48:                    *ret_val = -1;
49:            }
50:            __asm __volatile ("sti" ::);
51:            return 0;
52:    }
53:    #endif
```

그림 3.9: 카운터 값을 응용 프로그램으로 전달하기 위한 시스템콜

제 4 장

UNIX 운영체제에서 TCP/IP 성능 측정

제 1 절 BSD UNIX

UNIX 운영체제는 대표적인 다중 사용자(multi-user) 운영체제로 MIT에 의해 개발된 MULTICS를 기반으로 하여 개발되었다[7]. 처음에는 PDP-11 미니컴퓨터에 장착되었으나 현재는 슈퍼 컴퓨터를 비롯하여 개인용 컴퓨터에 이르기까지 거의 모든 프로세서 및 컴퓨터에 탑재되고 있으며[7], 대표적인 배포판으로는 AT&T UNIX System V, Microsoft Xenix System V, BSD(Berkeley Software Distribution) 등이 있다. 각각은 사용자 프로그램에는 일관된 인터페이스를 제공하나 커널 내부 및 컴퓨터 하드웨어에 대해서는 나름대로의 특징적인 서비스를 제공한다. AT&T UNIX System V와 Microsoft Xenix System V는 주로 상업적인 UNIX 배포판 및 커널(kernel core)을 일컬으며, BSD는 연구 목적으로 UCB(University of California, Berkeley)에서 제공하는 UNIX 및 커널(kernel core)을 의미한다.

BSD는 UCB의 CSRG(Computer Systems Research Group)으로부터 배포가 되고 있다. 1982년과 1983년 사이 4.1cBSD, 4.2BSD가 배포되었으며, 이들 배포판

의 가장 큰 특징은 UNIX 상에 대부분의 통신 프로토콜을 구현한 것이다. 특히 DARPA Internet Protocol인 TCP/IP를 구현하고 당시의 거의 모든 네트워크 디바이스에 대한 지원을 한 것은 UNIX와 TCP/IP를 널리 보급하는데 기여를 하였다. 이후, 4.3BSD, 4.4BSD가 발표되어 TCP/IP 및 커널의 내부 성능이 향상되면서 오늘에 이르고 있다. BSD가 UNIX 및 TCP/IP에 대해 중요한 의미를 가지는 것은 전술한 바와 같이 네트워크 프로토콜에 대한 축적된 기술에 있다. 많은 운영체제들이 BSD를 기준으로 TCP/IP를 제작하고 있으며 TCP/IP에 관한 대부분의 연구들이 BSD를 기반으로 행해지고 있기 때문이다. 이에 본 논문은 4.4BSD를 기반으로 하여 IBM PC에 구현이 된 FreeBSD를 운영체제로 사용하였으며 이하 UNIX라 하겠다.

제 2 절 실험 환경

본 절에서는 UNIX 운영체제 하에서 TCP/IP의 데이터 전송/수신 수행 시간과 메모리 대역폭 요구 사항을 측정하였다. 통신 노드간의 연결 설정/해제에도 많은 시간이 소요되나 멀티미디어 데이터와 같은 대량의 데이터를 전송할 경우 그 빈도수가 적으며 수행 시간이 각 노드의 상태에 의존적이므로 이 부분은 고려하지 않았다. 즉, 실제 상황에서 TCP/IP의 평균 처리율이 아닌 TCP/IP 자체의 처리 능력을 평가하기 위하여 두 대의 호스트 컴퓨터 사이에서 TCP 연결을 설정한 후 여러 가지 크기의 데이터를 전송/수신할 때의 수행 시간을 측정하였다. TCP가 한 번에 처리할 수 있는 데이터의 크기(세그먼트 크기)는 제한이 없으나 데이터링크 계층의 MTU(Maximum Transmission Unit)보다 클 경우, IP에 의해 단편화/재조립(fragmentation/reassembly)등의 복잡한 과정이 수행되게 되므로, 최대 크기는 보통 MTU로 설정된다. 실험에 쓰인 데이터의 크기는 100 바이트, 300 바

이트, 500 바이트, 1440 바이트로 전송 및 수신이 한 세그먼트에 의해 끝나도록 하였다. 특히 데이터의 크기가 1440 바이트일 경우 IP 데이터그램의 크기는 TCP 헤더 40 바이트(TCP 헤더 20 바이트 + 옵션 20 바이트), IP 헤더 20 바이트를 포함하여 1500 바이트가 되므로 이더넷에 설정된 MTU와 일치하게 된다.

그림4.1, 그림4.2는 TCP/IP의 데이터 전송/수신 성능을 측정하기 위한 실험 환경을 개념적으로 나타낸 것이다. 그림의 괄호 안에 있는 숫자는 성능 측정이 수행된 부분을 나타낸 것이며 구체적인 사항은 4, 5절에서 설명된다. 두 대의 개인용 컴퓨터를 10BaseT 라인으로 연결하였으며 운영체제는 FreeBSD(Release 2.1, 4.4BSD)를 사용하였다. CPU는 캐쉬 크기가 256 킬로 바이트인 인텔 펜티엄 프로세서(166MHz)를 사용하였으며, 마더보드(motherboard)는 PCI(Peripheral Component Interconnect), ISA(Industry Standard Architecture) 버스(bus)를 지원하는 Micronics 54Hi+를 사용하였다. 여기에 3Com ISA 이더넷 콘트롤러와 PCI Wide-SCSI(Small Computer Systems Interface) 콘트롤러 및 하드 디스크를 탑재하였다.

제 3 절 성능 측정 및 평가 방법

3.1 성능 측정 방법

성능 측정은 2.2절에서 기술한 드라이버를 이용하였다. TCP/IP 수행 모듈의 처리 시간을 측정하기 위해 TSC를 사용하였고, MSR의 이벤트 중 여덟 가지 이벤트를 사용하였다. 측정한 이벤트는 Data read, Data write, Instructions executed, Hardware interrupts, Data TLB (translation look-aside buffer) miss, Data read miss, Data write miss, Code cache miss이며, 측정 대상이 되는 TCP/IP 코드가 UNIX의 커널에 구현되어 있으므로 이들 이벤트의 측정은 커널 모드(kernel

그림 4.1: TCP/IP 전송 성능을 측정하기 위한 실험 환경

그림 4.2: TCP/IP 수신 성능을 측정하기 위한 실험 환경

mode)상에서 수행하였다. 이 때, Data read는 프로그램이 수행되는 동안 일어난 메모리 읽기의 수이고, Data writes은 메모리 쓰기의 수이다. 수행된 명령어의 총 수는 Instructions executed에 나타나며, Hardware interrupts는 프로그램이 수행되는 도중 발생한 하드웨어 인터럽트의 수이다. Data TLB(Translation-Lookaside Buffer) miss, Code TLB miss는 데이터 및 코드의 TLB에 대한 참조 실패(miss)의 수를 나타낸다. Data read miss, Data write miss, Code cache miss는 data cache에 대한 읽기/쓰기 실패와, code cache에 대한 참조 실패를 각각 나타낸다.

측정 예로, 그림 4.3은 체크섬의 성능을 측정하기 위한 코드를 나타낸다. 원래의 프로그램 코드에 22–23, 25, 48행을 추가하였다. 22, 23행은 디바이스 드라이버의 오버헤드를 측정하기 위한 부분으로, 카운터를 시작하고 정지하는 데 걸리는 시간을 측정할 수 있다. 25행은 체크섬의 성능 측정을 시작하는 부분이다. 환경 변수로 체크섬을 수행할 데이터의 크기를 설정한 후 측정을 시작하였고, 체크섬의 수행이 끝나는 48행 부분에서 카운터의 값을 읽었다. 22, 25행이 나타내는 것처럼, 드라이버의 오버헤드는 0번째 저장 장소에 저장되고, 체크섬의 수행 성능은 1번째에 저장되게 된다.

그림 4.4, 4.5는 카운터에 대한 제어를 수행하거나, 커널에 저장된 측정 결과를 응용프로그램을 전달하기 위한 목적으로 제작한 프로그램이다. 5행은 이벤트 카운터를 커널 모드로 설정하기 위해 필요한 비트 표현($40_{(16)} = 1000000_{(2)}$)이고, 6행은 현재 설정된 측정 이벤트를 알아내는데 필요한 비트 표현이다($3F_{(16)} = 111111_{(2)}$). 37행의 배열은 커널에 저장된 측정 결과를 저장하게 될 응용프로그램의 저장 장소이다. 커널에 설정된 크기와 똑 같이 10개를 저장할 크기를 가지게 된다. 그림 4.5는 카운터에 대한 실제 제어를 수행하는 부분들이다. 1–10행의 코드는 커널에 저장되어 있는 측정 결과를 배열 '`dmp`'에 저장하고 출력해 주는 부분이다. 11–15행

```
 1:    #include <sys/param.h>
 2:    #include <sys/systm.h>
 3:    #include <sys/mbuf.h>
 4:
 5:    #ifdef PCNT
 6:    #include <machine/pcnt_proto.h>
 7:    #endif
 8:
 9:    /* omitted some lines intentionally */
10:
11:    int
12:    in_cksum(m, len)
13:     register struct mbuf *m;
14:     register int len;
15:    {
16:     register u_short *w;
17:     register unsigned sum = 0;
18:     register int mlen = 0;
19:     int byte_swapped = 0;
20:     union { char c[2]; u_short s; } su;
21:
22:     start_cnt(NULL, 0);
23:     stop_cnt(NULL);
24:
25:     start_cnt(1, len);
26:
27:     for (;m && len; m = m->m_next) {
28:     if (m->m_len == 0)
29:     continue;
30:     w = mtod(m, u_short *);
31:     if (mlen == -1) {
32:     /*
33:      * The first byte of this mbuf is the continuation
34:      * of a word spanning between this mbuf and the
35:      * last mbuf.
36:      */
37:
38:     /* su.c[0] is already saved when scanning previous
39:      * mbuf.  sum was REDUCEd when we found mlen == -1
40:      * branches &c small.
41:      */
42:
43:     /* omitted some lines intentionally */
44:
45:     }
46:     REDUCE;
47:
48:     stop_cnt(1);
49:
50:     return (~sum & 0xffff);
51:    }
```

그림 4.3: 체크섬의 성능을 측정하기 위한 커널 상의 프로그램 코드

```
 1:    #include <stdio.h>
 2:    #include <sys/syscall.h>
 3:    #include <machine/pcnt_var.h>
 4:
 5:    #define KERNEL_ACCESS        0x40
 6:    #define EV_MASK 0x003F
 7:
 8:            char           *pname;
 9:
10:            char           *status[] = {
11:             "FAIL",
12:             "INIT",
13:             " RNG",
14:             "SET"
15:            };
16:
17:            char           *event_name[] = {
18:             "Data read",/* 0 */
19:             "Data write",
20:             "Data TLB miss",
21:             "Data read miss",
22:             "Data write miss",
23:    /* omitted some lines intentionally */
24:             "Breakpoint 3 match",
25:             "Hardware interrupts",
26:             "Data read or data write",/* 0x28 */
27:             "Data read miss or data write miss",
28:            };
29:
30:
31:    int
32:    main(argc, argv)
33:            int             argc;
34:            char           *argv[];
35:    {
36:
37:            struct cnt_pool dmp[MAX_POOL];
38:            unsigned int    flag;
39:            unsigned short  flag0, flag1;
40:            int             i;
41:
42:            pname = argv[0];
```

그림 4.4: 측정 결과를 얻어내기 위한 제어 프로그램의 예(전반부)

```
 1:     if (argc == 2 && strcmp("dump", argv[1]) == 0) {
 2:     syscall(CALL_NO, DUMP, dmp);
 3:     printf("\nCurrent values of counter buffer\n\n");
 4:     for (i = 0; i < MAX_POOL; i++)
 5:     printf("[%2d:%4s:%5d]{:%10d:%10d:%10d:}\n", \
 6:            i, status[dmp[i].status], dmp[i].para, \
 7:            dmp[i].tsc, dmp[i].msr0, dmp[i].msr1);
 8:     printf("\n");
 9:     return 0;
10:     }
11:     if (argc == 2 && strcmp("clear", argv[1]) == 0) {
12:     syscall(CALL_NO, CLEAR);
13:     printf("All entries are in initial state!\n");
14:     return 0;
15:     }
16:     if (argc == 3) {
17:
18:     flag0 = (unsigned short) strtol(argv[1], NULL, 16);
19:     flag1 = (unsigned short) strtol(argv[2], NULL, 16);
20:
21:     if (event_name[flag0] == NULL || \
22:         event_name[flag1] == NULL) {
23:     fprintf(stderr, "%s:invalid event type 0x%x:0x%x\n", \
24:     pname, flag0, flag1);
25:     return -1;
26:     }
27:     flag = (flag1 | KERNEL_ACCESS) << 16;
28:     flag |= flag0 | KERNEL_ACCESS;
29:
30:     syscall(CALL_NO, SET_EVENT, &dmp, &flag);
31:     }
32:     syscall(CALL_NO, GET_EVENT, &dmp, &flag);
33:     flag0 = flag & EV_MASK;
34:     flag1 = (flag >> 16) & EV_MASK;
35:
36:     printf("Current state of event counter\n");
37:     printf("0x%x: %s\n", flag0, event_name[flag0]);
38:     printf("0x%x: %s\n", flag1, event_name[flag1]);
39:
40:     return 0;
41:
42:    }
```

그림 4.5: 측정 결과를 얻어내기 위한 제어 프로그램의 예(후반부)

은 성능 측정을 시작할 수 있도록 커널 내의 저장 장소를 초기화 하는 부분이다. 16–31행은 이벤트 카운터의 측정 이벤트를 설정하는 부분으로, MSR 17번에 전달하기 위한 비트 표현을 생성하고(18–19행, 27–28행) 시스템콜을 이용해 이벤트를 설정하는(30행) 부분이다. 32–38행은 이벤트 카운터의 현재 설정 상태를 얻어내는 부분이다. MSR 17번으로부터 얻은 결과 중 이벤트의 종류를 나타내는 하위 6비트를 추출한 후(33–34행) 해당 이벤트를 표현하기 위한 적당한 문자열을 화면에 출력하도록 하고 있다(36–38행).

```
 1:    [mars:/home/hpcclab/tcpip/.driver/cnt] cnt_ctrl 0 1
 2:    Current state of event counter
 3:    0x0: Data read
 4:    0x1: Data write
 5:    [mars:/home/hpcclab/tcpip/.driver/cnt] cnt_ctrl clear
 6:    All entries are in initial state!
 7:    [mars:/home/hpcclab/tcpip/.driver/cnt] cnt_ctrl dump
 8:
 9:    Current values of counter buffer
10:
11:    [ 0:INIT:    0]{:         0:         0:         0:}
12:    [ 1:INIT:    0]{:         0:         0:         0:}
13:    [ 2:INIT:    0]{:         0:         0:         0:}
14:    [ 3:INIT:    0]{:         0:         0:         0:}
15:    [ 4:INIT:    0]{:         0:         0:         0:}
16:    [ 5:INIT:    0]{:         0:         0:         0:}
17:    [ 6:INIT:    0]{:         0:         0:         0:}
18:    [ 7:INIT:    0]{:         0:         0:         0:}
19:    [ 8:INIT:    0]{:         0:         0:         0:}
20:    [ 9:INIT:    0]{:         0:         0:         0:}
21:
22:    [mars:/home/hpcclab/tcpip/.driver/cnt] _
```

그림 4.6: 제어 프로그램을 이용한 측정 초기화 작업

그림 4.6은 측정을 수행하기 전의 초기화 과정을 실제로 나타낸 것이다. 1–4행은 제어 프로그램을 이용해 측정 이벤트를 설정하는 단계이다. 3–4행이 나타내는 것처럼 이벤트는 메모리 읽기/쓰기로 설정하였다. 5행은 커널 상의 카운터 값 저

```
 1:    [mars:/home/hpcclab/tcpip/.driver/cnt] cnt_ctrl dump
 2:
 3:    Current values of counter buffer
 4:
 5:    [ 0: SET:    0]{:       159:         4:         8:}
 6:    [ 1: SET:   20]{:       797:        23:        10:}
 7:    [ 2:INIT:    0]{:         0:         0:         0:}
 8:    [ 3:INIT:    0]{:         0:         0:         0:}
 9:    [ 4:INIT:    0]{:         0:         0:         0:}
10:    [ 5:INIT:    0]{:         0:         0:         0:}
11:    [ 6:INIT:    0]{:         0:         0:         0:}
12:    [ 7:INIT:    0]{:         0:         0:         0:}
13:    [ 8:INIT:    0]{:         0:         0:         0:}
14:    [ 9:INIT:    0]{:         0:         0:         0:}
15:
16:    [mars:/home/hpcclab/tcpip/.driver/cnt] _
```

그림 4.7: 제어 프로그램을 이용한 측정 결과 출력

장 영역을 초기화 하는 부분이며, 초기화 여부를 확인하기 위해 7행의 명령어 수행을 통해 저장 영역의 값을 확인하였다. 11-20행이 나타내는 것처럼 저장 영역의 모든 값이 초기 상태(INIT)와 '0'으로 설정되었다. 그림 4.7은 적절한 실험 환경을 통해 성능 측정을 수행한 후, 결과 값을 확인하는 방법을 보이고 있다. 제어 프로그램을 이용해 저장 영역의 내용을 출력하면, 5–6행과 같이 측정 결과가 보여지게 된다. 대괄호로([ ]) 표현된 부분은 측정에 관련된 정보이다. 처음 부분의 숫자는 저장 영역의 번호이며, 두 번째는 측정 결과에 대한 정보이다. 'SET'으로 나타나는 것은 측정이 올바로 완료되었음을 의미한다. 세 번째의 숫자는 측정 당시의 환경에 대한 정보로, 숫자 '20'은 체크섬을 수행한 데이터의 크기가 20이었음을 나타낸다. 다음으로 중괄호({ })로 나타나는 부분의 숫자는 각 카운터의 결과 값이다. 처음 부분은 싸이클 카운터의 값이며, 나머지 두 개는 이벤트 카운터의 값을 나타낸다.

측정 결과로부터 해당 모듈의 성능을 산출하는 방법은 다음과 같다. 그림 4.7의

6행에 나타난 측정 결과는 드라이버를 구동하는데 소요된 오버헤드를 포함하므로, 체크섬의 순수한 성능을 산출하기 위해서는 측정 결과에서 드라이버의 오버헤드를 빼야 한다. 따라서 예로 보인 체크섬의 수행 성능은 싸이클 수가 638(797–159), 메모리 읽기의 수가 19(23–4), 메모리 쓰기의 수가 2(10–8)가 된다. 이와 같이, 성능 산출은 실제 측정 결과에서 드라이버의 오버헤드를 빼는 과정을 통해 얻어지며, 본 논문에서 제시하는 모든 측정 결과는 이같은 방법으로 얻어진 것이다.

3.2 성능 평가 방법

측정한 싸이클 수 및 이벤트 중 논문에서 성능 평가 기준으로 선택한 것은 TCP/IP 모듈이 수행되는 동안의 싸이클 수, 수행된 명령어 수, 메모리 참조 수(읽기, 쓰기의 횟수를 합한 값)이다. 싸이클 수와 명령어 수는 프로그램의 수행 시간과 복잡도를 나타내는 척도가 되며, 메모리 참조의 수는 메모리 대역폭을 계산하기 위한 기초 자료가 된다. 다만 인스트럭션 수에 대한 성능 평가 시에는 다음과 같은 경우에 유의하여야 한다. 성능평가가 수행된 부분의 코드가 에셈블리의 'REP' 접두사를 사용하여 수행된 경우로, 이 경우 이벤트 카운터의 값은 '1' 만큼 증가하게 된다. 예를 들면, 'REP'접두사를 사용하여 메모리 읽기를 10 번 수행한 경우, 수행된 명령어 수는 '1' 만큼 증가하고 메모리 읽기 수는 '10' 만큼 증가하게 된다.

측정 결과는 시스템의 상태에 따라 약간씩 다른 값을 가질 수 있는데, 예를 들면 펜티엄 프로세서 내의 인터럽트 상태나 캐쉬(cache)의 상태에 따라 수행되는 시간 및 명령어 수, 메모리 참조의 횟수가 차이가 날 수 있다. 따라서, 일정한 조건하에서 TCP/IP의 성능을 산출하기 위하여 다음과 같은 방법을 사용하였다.

① 하드웨어 인터럽트가 발행하였을 경우의 측정치는 고려하지 않음

② TLB misses(Code, Data) 수가 원하는 범위의 편차를 벗어난 경우의 측정치

는 고려하지 않음

③ Cache misses(Data read/write, code) 수가 원하는 범위의 편차를 벗어난 경우의 측정치는 고려하지 않음

④ ①, ②, ③과 같은 방법으로 얻은 20개의 측정치에 대해 평균을 구함

제 4 절 TCP/IP 전송 성능 측정

그림 4.1에서 보는 바와 같이 TCP 연결을 설정한 후에, 사용자 프로그램이 데이터를 전송하기 시작하는 시점에서부터 종료하는 시점 사이의 각 모듈의 수행 시간을 측정하였다. 전송하는 부분은 크게 데이터를 전송하는 부분과, 전송된 데이터에 대한 ACK 세그먼트를 받아 처리하는 과정으로 나누어 볼 수 있다. 먼저 데이터를 전송하는 과정은 다음과 같다. 사용자 데이터를 사용자 영역으로부터 커널 영역으로 복사하기 위해 메모리를 할당하고 데이터를 복사한다(1). 이 데이터를 TCP로 보낸다. TCP는 옵션 처리, TCP 체크섬 등을 수행하고, 생성된 세그먼트를 IP로 보낸다(2). IP는 옵션 처리, 주소 설정을 하고, IP 체크섬을 수행한 후 데이터그램을 네트워크 디바이스로 보낸다(3). 네트워크 디바이스는 물리 계층으로 보낼 프레임을 형성하고, 이를 전송 매체로 보내게 된다(4).

데이터를 전달받은 수신 측은 데이터에 이상이 없을 경우 데이터에 대한 ACK을 전송하게 되고, 전송 측은 이 ACK 세그먼트를 처리하게 된다. ACK 세그먼트를 처리하기 위해 디바이스 드라이버는 전송 매체로부터 ACK 세그먼트가 포함된 프레임을 읽어 들인다. 오류가 없을 경우 프레임으로부터 IP 데이터그램을 추출해 내며, 이를 IP 영역으로 전달하게 된다(실제로는 소프트웨어 인터럽트에 의해 수행이 개시됨)(5). IP는 데이터그램에 대해 체크섬을 수행하고 오류가 없으면 데

이터그램을 TCP로 전달한다(6). TCP는 ACK 세그먼트에 대한 처리를 하고, 전송한 데이터의 메모리 영역을 해제한다(7).

제 5 절 TCP/IP 수신 성능 측정

그림4.2에서 보는 바와 같이 TCP 연결을 설정한 후에 사용자 프로그램이 데이터를 수신하기 시작하는 시점에서부터 종료하는 시점 사이의 시간을 측정하였다. 수신하는 과정은 데이터를 전달받는 부분과, 전달받은 데이터에 대한 ACK 세그먼트를 형성하여 전송하는 부분으로 나눌 수 있다. 데이터를 수신하는 과정은 다음과 같다. 전송 측의 프레임이 네트워크 드라이버에 도착하면 네트워크 드라이버는 프레임으로부터 IP 데이터그램을 추출하여 IP로 보낸다(1). IP는 데이터그램으로부터 주소 확인과 체크섬 계산을 한다. 오류가 없을 경우 이를 TCP로 보낸다(2). TCP는 데이터를 전달받아 체크섬을 계산한 후 오류가 없으면 이를 소켓 버퍼에 추가(socket buffer append)하고 소켓을 wakeup 한다(3). Wakeup에 의해 활성화된 소켓은 소켓 버퍼로 전달된 데이터를 사용자 영역으로 복사하게 된다(4). 이와 같이 데이터를 전달받은 수신 측은 수신한 데이터에 대한 ACK 세그먼트를 형성하여 이를 전송 측에 보내게 된다. ACK 세그먼트를 보내는 과정은 TCP 전송 과정과 같다. ACK 세그먼트는 헤더 이외의 데이터를 포함하고 있지 않으므로 곧바로 TCP는 ACK 세그먼트를 형성하고(5), 이를 IP(6), 네트워크(7)를 통해 전송 측에 전달한다.

제 5 장

실험 결과 및 분석

그림 5.1, 5.2, 5.3은 그림 4.1과 같은 환경에서의 TCP/IP 전송 성능을 나타낸 것이다. 100, 300, 500, 1440 바이트의 데이터를 보내고 ACK 세그먼트를 받을 때까지의 각 모듈의 수행 싸이클 수, 명령어 수, 메모리 참조 수를 나타내었다. 그림 5.1이 나타내는 것처럼 전송 시간의 대부분은 데이터링크 계층인 이더넷에서 소비된다. 이는 상대적으로 저속인 이더넷의 처리 능력(10 Mbps) 때문이다. 또한 ACK 세그먼트 처리에도 비교적 많은 시간이 소모되는데, 이유는 옵션 처리 부분이 포함되어 있고 ACK 처리 루틴이 헤더 예측 알고리즘(header prediction algorithm)[12]에 의해 처리되지 않았기 때문이다. 전송 성능을 한 세그먼트 최대 전송량인 1440 바이트에 대해 이더넷의 지연 시간을 고려하지 않고 계산해 보면, 데이터를 전송하고 ACK 세그먼트를 처리하는데 0.18 ms의 시간이 소요되어서 TCP/IP의 데이터 송신부의 처리 속도는 약 62 Mbps가 된다. 메모리 참조가 가장 많이 일어나는 부분은 TCP output 부분으로 10111 싸이클 동안 1029 번의 메모리 참조가 일어났다. 4 바이트 단위로 메모리 참조가 일어난다고 가정하면 이 부분의 메모리 대역폭은 약 68 Mbytes/sec가 된다. 벌크 데이터(bulk data) 전송에 대해 ACK 세그먼트를 처리하는 시간이 데이터 전송에 소모되는 시간에 비해 충분히 작다고 가정

하면 TCP/IP 전송 시간은 데이터를 전송하는데 소요되는 시간이 될 것이다. 이와 같은 상황을 가정하여 TCP/IP 전송 능력을 계산해 보면 수행시간이 0.12 ms가 되어서 처리 속도는 약 98 Mbps가 된다. 그리고 이 수치는 TCP/IP 전송 처리의 상한선이 될 것이다.

그림 5.1: TCP/IP 데이터 전송 시의 싸이클 수

그림 5.4, 5.5, 5.6은 그림 4.2와 같은 환경에서의 TCP/IP 수신 성능을 나타낸 것이다. 그림 5.4가 나타내는 것처럼 수신된 데이터를 사용자 영역으로 복사는 과정이 가장 많은 비율을 차지하였고, 이더넷을 통해 데이터를 전송/수신하는 부분이 긴 수행 시간을 나타내었다. TCP 수신 부분이 비교적 작은 수치를 보였는데,

그림 5.2: TCP/IP 데이터 전송 시의 인스트럭션 수

그림 5.3: TCP/IP 데이터 전송 시의 메모리 참조 수

이 부분은 헤더 예측 알고리즘에 의해 처리되었기 때문이다. 수신 성능을 한 세그먼트 최대 수신량인 1440 바이트에 대해 이더넷의 지연시간을 고려하지 않고 계산해 보면, 데이터를 수신하고 ACK 세그먼트를 전송하는데 0.17 ms의 시간이 소요되어서 TCP/IP의 데이터 수신부의 처리 속도는 약 68 Mbps가 된다. 메모리 참조가 가장 많이 일어나는 경우는 커널 영역에서 사용자 영역으로 데이터를 복사하는 부분으로 10986 사이클 동안 3148 번의 메모리 참조가 일어났다. 4 바이트 단위로 메모리 참조가 일어난다고 가정하면 이 부분의 메모리 대역폭은 약 190 Mbytes/sec가 된다. ACK 세그먼트를 전송하는 시간을 고려하지 않고 순수하게 데이터를 수신하는데 소요되는 시간은 0.12 ms가 되어서 수신 속도는 약 94 Mbps가 된다. 그리고 이 수치는 TCP/IP 수신 처리의 상한선이 될 것이다.

실험 결과에 비추어 TCP/IP의 병목 부분은 크게 두 가지로 나누어 볼 수 있다. 네트워크 디바이스 드라이버 부분과 운영체제/소켓 간의 데이터 복사 부분이 될 것이다. 네트워크 디바이스의 병목 현상은 본 실험에서 사용한 이더넷의 상대적으로 낮은 속도 때문으로 ATM과 같은 고속 네트워크를 사용하면 어느 정도 해결할 수 있다. 운영체제와 소켓간의 데이터 복사에 대한 해결책으로는 자료 복사가 일어나지 않게 하는 제안 및 구현이 이루어진 바 있다[13, 16]. 또한, 참고문헌[35]와 같이 TCP/IP를 위한 독립적인 하드웨어를 설계한다면 처리 능력은 충분히 향상될 것으로 보인다.

그림 5.4: TCP/IP 데이터 수신 시의 싸이클 수

그림 5.5: TCP/IP 데이터 수신 시의 인스트럭션 수

그림 5.6: TCP/IP 데이터 수신 시의 메모리 참조 수

제 6 장

결론

본 논문에서는 TCP/IP의 데이터 전송/수신 성능을 펜티엄 166 MHz CPU와 UNIX 운영체제를 탑재한 컴퓨터 상에서 측정 및 분석하였다. TCP 연결이 설정된 이후에 한 세그먼트를 전송/수신하는 실험을 하여, 각 모듈이 수행되는데 소요되는 시간을 측정하였다. 측정 도구는 모듈이 수행되는 동안의 싸이클 수, 명령어 수, 메모리 참조 수 등을 산출할 수 있는 인텔 펜티엄 프로세서 내의 카운터들을 이용하였으며, 이들 카운터의 결과 값으로부터 모듈의 수행 시간, 복잡도, 메모리 대역폭을 계산하였다.

실험 결과, TCP/IP 처리의 병목 부분은 TCP/IP 전송의 경우 네트워크 디바이스가 가장 큰 병목 부분이었으며, 다음으로 사용자 영역과 커널간의 데이터 복사, 체크섬을 포함한 TCP 전송 처리 순으로 나타났다. TCP/IP 수신의 가장 큰 병목 부분은 커널 영역과 사용자 영역간의 데이터 복사 부분으로 나타났고, 다음으로는 네트워크 디바이스 처리, 체크섬을 포함한 TCP 수신부로 나타났다.

저서 목록

[1] A. K. Agrawala and D. Sanghi. Design and evaluation of an adaptive flow control scheme. *Eleventh Annual Joint Conference of the IEEE Computer and Communications Societies*, 3:2391–2397, 1992.

[2] D. Anderson and T. Shanley. *Pentium Processor System Architecture*. Mindshare Press, 1993.

[3] T. Berners-Lee and et. al. The World-Wide Web. *Communications of the ACM*, 37(8), Aug. 1994.

[4] J. Bradley Chen and et al. The Measured Performance of Personal Computer Operating Systems. *ACM Trans. Computer Systems*, Feb. 1996.

[5] D. D. Clack and V. Jacobson. An analysis of TCP processing overhead. *IEEE Commun.*, 27:23–29, June 1989.

[6] D. D. Clark. Window and Acknowledgment Strategy in TCP. *RFC 813*, July 1982.

[7] Silberschatz Galvin. *Operating System Concepts*. Addison-Wesley, 1994.

[8] G. Held. *Ethernet networks : design, implementation, operation, and management*. John Wiley & Sons, 1994.

[9] Jau-Hsiung Huang and Chi-Wen Chen. On Performance Measurements of TCP/IP and its Device Driver. *IEEE Proc. of 17th Conference on Local Computer Networks*, pages 568–575, 1992.

[10] Intel Corporation. *Pentium Family User's Manual:Architecture and Programming Manual*, volume 3. Intel Corporation, 1994.

[11] V. Jacobson. Congestion Avoidance and Control. *Computer Communication Review*, 18(4):314–329, Aug. 1988.

[12] V. Jacobson. 4BSD TCP Header Prediction. *Computer Commun. Review*, 20(2):13–15, Apr. 1990.

[13] V. Jacobson. Efficient protocol implementations. *ACM SIGCOMM 90 Tutorial*, Sep. 1990.

[14] J. Kay and J. Pasquale. The Importance of Non-data Touching Processing Overheads in TCP/IP. *ACM SIGCOMM Computer Communication Review*, 23:259–268, Oct. 1993.

[15] J. Kay and J. Pasquale. Profiling and Reducing Processing Overheads in TCP/IP. *IEEE/ACM transactions on networking*, 4(6):817–828, Dec. 1996.

[16] Hsiao keng Jerry Chu. Zero-Copy TCP in Solaris. *Proc. of the USENIX 1996 Annual technical conference*, pages 253–264, 1996.

[17] Hyok Kim, Hongki Sung, and Hoonbock Lee. Performance Analysis and Feasibility Study of a Parallelized TCP/IP Implementations. *Proc. of Korean Inst. of Comm. Sciences Conference*, 15(2):443–446, Nov. 1996.

[18] Hyok Kim, Hongki Sung, and Hoonbock Lee. Performance Analysis of the TCP/IP Protocol Under UNIX Operating Systems for High Performance Computing and Communications. *Proc. of High Performance Computing ASIA '97*, pages 499–504, May 1997.

[19] O. G. Koufopavlou, A. N. Tantawy, and et.al. Parallel TCP for High Performance Communication Subsystems. *Proc. of 17th IEEE Global Telecommunications Conference*, pages 1395–1399, Dec. 1992.

[20] O. G. Koufopavlou, A. N. Tantawy, and M. Zitterbart. Analysis of TCP/IP for High Performance Parallel Implementations. *Proc. of 17th IEEE conference on Local Computer Networks*, pages 576–585, Sep. 1992.

[21] D. C. Lynch. *Internet Handbook*. Addison-Wesley, 1993.

[22] M. H. MacDougall. *Simulating Computer Systems:Techniques and Tools*. MIT Press, 1987.

[23] J. Martins and J. P. Hubaux. Evaluating Performance from Formal Specification. *Proc. of IEEE MASCOTS 96*, pages 285–290, 1996.

[24] T. Mathisen. Pentium secrets. *Byte*, pages 191–192, July 1994.

[25] D. E. McDysan and D. L. Spohn. *ATM:Theory and Application*. McGraw-Hill, 1994.

[26] D. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[27] A. Mink and et. al. Hardware Measurement Techniques for High-speed Networks. *Journal of High Speed Networks*, 3(2):187–207, 1994.

[28] A. N. Netravali, W. D. Roome, and K. Sabnani. Design and Implementation of a High-Speed Transport Protocol. *IEEE Trans. on Commun.*, 38(11):2010–2024, Nov. 1990.

[29] Thomas F. La Porta and Mischa Schwartz. Architectures, Features and Implementation of High-Speed Transport Protocols. *IEEE Network Magazine*, pages 14–22, May 1991.

[30] J. B. Postel. Internet Protocol. *RFC 791*, Sep. 1981.

[31] J. B. Postel. Transmission Control Protocol. *RFC 793*, Sep. 1981.

[32] M. Schmit. Optimizing Pentium Code. *Dr. Dobb's Journal*, Jan. 1994.

[33] W. R. Stevens. *TCP/IP Illustrated:The Protocols*. Addison-Wesley, 1994.

[34] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, second edition, 1989.

[35] Dalton G. Waston and et al. Afterburner (network-independent card for protocols). *IEEE Network*, 7(36-43), July 1993.

[36] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated:The Implementation*. Addison-Wesley, 1995.

Performance Analysis of TCP/IP Data Send/Receive Processing Under UNIX Operating Systems

Hyok Kim

Department : Computer Engineering
Major : Computer Engineering
Graduate School, Hallym Univ

This paper analyzes the performance of TCP/IP processing in sending and receiving data under UNIX operating systems. We use the counters in the Intel Pentium Processor to measure the number of cycles, the number of instructions executed, and the number of memory accesses in data send/receive parts of the TCP/IP processing. Empirical results show that the major bottlenecks of TCP/IP processing are Ethernet processing and data copy from kernel space to user space in data sending and receiving, respectively.

목 차

그림 목차

표 목차

```
$Id: Thesis.tex,v 1.6 1998/01/14 03:06:43 hkim Exp hkim $

$Log: Thesis.tex,v $
Revision 1.6  1998/01/14 03:06:43  hkim
**********************
* READY TO BE BOUND *
**********************

Revision 1.5  1998/01/12 06:14:52  hkim
************************************************
* final final final final final final final *
************************************************

Revision 1.4  1998/01/11 12:43:38  hkim
* changed vertical space of tables (contents, list, figure)

Revision 1.3  1998/01/11 05:41:34  hkim
final draft
fitted to B5 papers
separated cover pages from main text
so, this document generates only text portion of my thesis

Revision 1.2  1998/01/10 10:36:54  hkim
revision with new format
intermediate version

Revision 1.1  1998/01/10 05:38:21  hkim
Initial revision
```