creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사 학위논문

TCP 프로토콜의 호스트 CPU 사용량 절감을 위한 TCP 오프로딩 하드웨어를 위한 소프트웨어 개발

2022 년 5 월

서울대학교 대학원

전기정보공학과

나성민

TCP 프로토콜의 호스트 CPU 사용량 절감을 위한 TCP 오프로딩 하드웨어를 위한 소프트웨어 개발

지도 교수 김장우

이 논문을 공학석사 학위논문으로 제출함
2022 년 8 월

서울대학교 대학원
전기정보공학부
나성민

나성민의 공학석사 학위논문을 인준함
2022 년 8 월

위 원 장 김재준 (인)

부위원장 김장우 (인)

위 원 심재웅 (인)

초 록

현대 서버의 높은 데이터 처리량과 낮은 레이턴시 소요를 만족시키기 위해 네트워크 속도가 점점 더 빨라지고 있다. 문제는 고성능의 네트워크에 TCP 프로토콜을 제공하려면 막대한 호스트 CPU 사용량이 필요하다는 것이다. 본 연구에서는 TCP 스택의 호스트 CPU 사용량을 프로파일링하여 문제의 심각성을 확인하고, 이에 대해서 기존 연구들이 제시하고 있는 해결책들을 검토하였다. 이를 통해서 TCP 스택을 하드웨어로의 오프로딩을 통해 호스트 CPU의 사용량을 줄이는 것이 미래의 수백 Gbps를 네트워크에 적용하기 가장 적절한 솔루션으로 판단하였다.

본 연구에서는 TCP 하드웨어 오프로딩 모델인 TOE(TCP Offloading Engine)를 제시하였다. 또한 애플리케이션이 TOE의 TCP기능을 사용할 수 있도록 중계해주는 소프트웨어인 TOE Manger를 설계 및 구현하였다. 이를 가장 대중적인 웹 서버인 NGINX에 패치해서 사용함으로써 엔드-투-엔드로 동작하는 것을 확인하였다. 또한 정량적인 평가를 통해, TOE에 TCP를 오프로딩 하면 Linux 커널의 TCP 스택을 사용하는 경우에 비해 동일 코어수로 2.5~3배 높은 성능을 달성하거나, 같은 성능을 2~4배 적은 코어로 달성하는 것을 확인하였다.

주요어 : TCP 오프로딩을 위한 소프트웨어 개발

학 번 : 2020-23372

목 차

제 1 장 서 론 1
제 1 절 연구의 배경 1
제 2 절 연구의 내용 3

제 2 장 TCP 스택 분석 5
제 1 절 TCP 프로토콜 5
제 2 절 TCP 프로토콜 스택의 연결 상태 관리 7
제 3 절 TCP 스택의 호스트 CPU 사용 9
제 4 절 선행 연구 분석 12

제 3 장 TCP 오프로딩 하드웨어 모델 14
제 1 절 TONIC의 TCP 오프로딩 하드웨어 14
제 2 절 기존 TONIC 하드웨어의 한계점 16
제 3 절 TOE 디자인 목표 18
제 4 절 TOE 아키텍처 20

제 4 장 TOE Manager 설계 및 구현 22
제 1 절 TOE 와의 인터페이스 22
제 2 절 TOE Manager 구현 26
제 3 절 멀티 코어로의 확장 28

제 5 장 평가 30
제 1 절 평가 환경 30
제 2 절 실험 결과 31

제 6 장 결론 및 제언 33

참고문헌 34

Abstract 39

그림 목차

[그림 1] 2000-2030 이더넷 로드맵 1

[그림 2] TCP 연결 상태(connection state) 7

[그림 3] iPerf의 처리량에 따른 호스트 CPU 사용량 (128byte 패킷) 10

[그림 4] TONIC의 하드웨어 오버뷰.....14

[그림 5] TONIC의 아키텍쳐 14

[그림 6] TOE(TCP offloading engine) 모델.....20

[그림 7] TOE Manager와 TOE 간의 인터페이스.....22

[그림 8] TOE Manager의 실제 동작.....26

[그림 9] Multi-core에서의 TOE Manager 29

[그림 10] Linux와 TOE에서 NGINX RPS 처리량.....31

[그림 11] Linux 와 TOE에서 CPU breakdown.....32

제 1 장 서 론

제 1 절 연구의 배경

현대 서버에서는 높은 데이터 처리량(throughput)과 낮은 레이턴시 (latency)로 데이터 송수신을 지원하는 것의 중요성이 커지고 있다. 현대 사회는 네트워크 이용자가 폭발적 증가와 그들이 요구하는 이미지나 동영상 같은 고화질 콘텐츠는 네트워크 트래픽(network traffic)의 크게 증가시켰다.[1] 또한 딥러닝(deep learning)[2], 마이크로서비스(microservice)[3,4]와 같이 서버 간의 통신을 많이 필요로 하는 워크로드들이 등장하기 시작하였다. 데이터 센터들은 이러한 네트워크 트래픽에 대한 증가한 수요를 만족시키기 위해서, 고성능의 네트워크 퍼포먼스를 가지는 서버들을 갖추는 것이 중요해졌다.

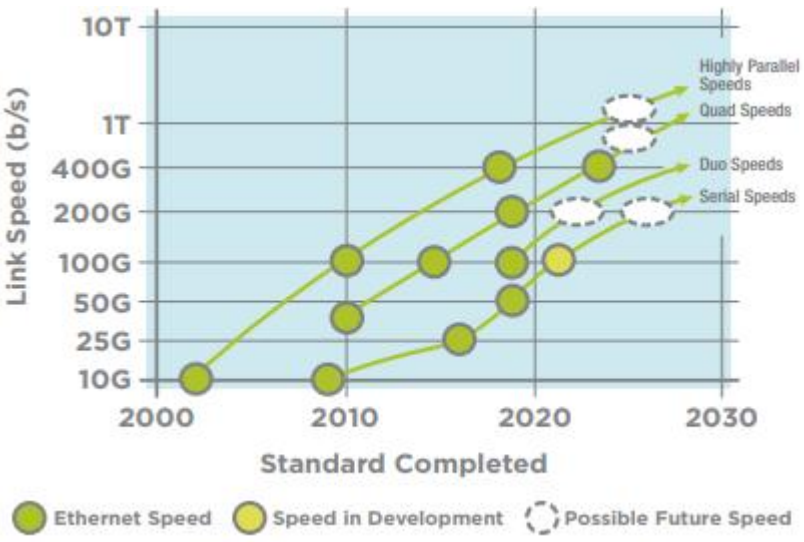

그림 1 2000-2030 이더넷 로드맵 [27]

이러한 높은 네트워크의 퍼포먼스에 대한 요구는 빠른 네트워크 링크의 개발로 이어졌다. 현재 많은 NIC 제조사들은 고속 네트워크

링크를 지원하는 NIC(Network Interface Card)를 새롭게 출 시 및 개발하고 있다. 그림 1은 이더넷 장치에 사용되는 링크 속도(link speed)의 로드맵을 나타난 것이다. 2004년의 링크 스피드는 1Gbps에 불과했지만, 2010년에 들어서는 10Gbps가 표준이 되었고, 2020년대에 들어와서는 100Gbps를 지원하는 NIC 8개를 활용해 단독으로 800Gbps 지원하는 서버가 등장하였다.[5] 앞으로 제조사들은 2024년까지 1,000Gbps를 지원하는 NIC 개발을 계획하고 있다.[6]

하지만 높은 네트워크 퍼포먼스를 달성하기 위해서는 네트워크 링크 속도를 높이는 것뿐만 아니라, 각 노드에 데이터의 컨트롤하는 네트워크 프로토콜을 빠르게 처리해야 한다. 각 노드가 네트워크 프로토콜을 빠르게 처리하지 못하면 네트워크 링크의 여유가 있어도 이를 채울만큼의 패킷을 충분히 생성할 수 없기 때문이다.

하지만 이러한 네트워크 프로토콜을 빠르게 처리하는 것은 방대한 호스트 CPU 사용량을 필요로 한다. 현재 네트워크는 안정적인 통신을 위해서 일반적으로 TCP 프로토콜을 사용하는데, 많은 논문에서 TCP 프로토콜을 위한 TCP 스택을 처리하는데 매우 많은 CPU 사이클을 소모한다는 연구 결과를 발표하고 있다.[7,8,9,10] 미래의 네트워크 속도가 빨라진다면, 이에 비례하여 네트워크 프로토콜이 더 많은 호스트 CPU 사이클을 소모하게 될 것이다. 이렇게 되면 호스트 CPU가 다른 애플리케이션을 처리할 시간이 줄어들게 되고, 최악의 경우 호스트 CPU 성능에 의해 네트워크 성능이 제한(bound)될 수 있다. 본 논문에서는 고성능의 데이터 링크에 대해서 TCP 프로토콜을 지원하면서도, 이로 인한 TCP 스택에 CPU 사이클을 줄이기 위한 방법을 제안하고 한다.

제 2 절 연구의 내용

본 연구는 TCP 스택의 호스트 CPU 점유가 얼마나 큰 문제인지 분석하기 위하여 프로파일링을 진행하였다. Linux 시스템에서 특정 속도의 데이터 링크를 포화시키는데 TCP 스택이 얼마만큼의 CPU 사이클 점유하는지 분석하였다. 실험 결과 100Gbps의 데이터 링크를 64, 128바이트의 페이로드(payload)를 갖는 패킷으로 가득 채우는데 각각 16, 32개의 CPU core를 사용해야 하는 것을 확인하였다. 미래의 네트워크 속도가 빨라져서, 데이터 링크의 속도가 800Gbps가 된다면, 이를 채우기 위해서 단순 계산으로도 128, 256개의 CPU 코어가 사용된다. 게다가 여러 코어에서 처리하는 경우 동기화(synchronization)을 위해서 각 코어 간의 통신, 공유자원에 대한 경쟁 등이 발생한다는 것을 생각하면 실제로는 위의 계산보다 훨씬 많은 호스트 CPU가 사용될 것이다.

이미 많은 선행 연구들이 이러한 각성을 예측하였고, 이에 대한 솔루션을 제시하였다. 본 연구에서는 선행 연구에서 TCP 스택의 호스트 CPU 사용량을 줄이기 위한 해결책을 3가지 방향으로 정리하였다. 첫 번째는 TCP 네트워크 스택 자체를 경량화(lightweight)하여 TCP 스택이 차지하는 CPU 사이클을 줄이는 방법이다. 두 번째는 스마트 닉(smart NIC)과 같이 NIC 내부의 디바이스 코어(device core)를 TCP 스택 수행에 활용하는 방식이다. 마지막은 FPGA나 ASIC을 활용하여 TCP만을 위한 하드웨어를 새롭게 설계하고, 이 하드웨어에 TCP 스택을 오프로딩(offloading)하는 방식이다.

본 연구에서는 미래의 수백 Gbps의 네트워크를 지원하기 위한

해결책 중 마지막으로 언급한 하드웨어에 오프로딩하는 방법이 가장 적합한 해결책이라고 판단하였다. 첫번째 방법의 경우, CPU의 제너럴한 아키텍쳐는 TCP 네트워크를 처리하는데 비효율적이며, 네트워크 스택을 개선한다 하더라도 그 영향이 제한적이어서, 장기적인 해결책으로 보기 어렵다고 판단하였다. 두번째 방법의 경우, NIC내부의 디바이스 CPU의 낮은 프로세싱 파워(processing power)로 인해서, 수백 Gbps의 네트워크에 TCP 프로토콜을 제공하는 것은 어려울 것으로 예측하였다. 마지막으로, FPGA 나 ASIC을 활용하여 TCP 스택을 위해 하드웨어를 설계하는 것은 높은 디자인 코스트를 필요로 한다. 하지만 TCP연산에 맞추어 하드웨어를 효율적으로 설계할 수 있고, 필요에 따라서 연산 유닛의 수를 늘려서 수백 Gbps의 네트워크에 필요한 연산 능력을 제공할 수 있어서 가장 잠재력이 높은 해결책으로 판단하였다.

본 연구에서는 하드웨어에 오프로딩 하는 방법을 문제의 해결책으로 제안하고자 한다. 이를 위해 먼저 기존의 연구들의 하드웨어 구현 방식과 한계점을 분석하여 새로운 하드웨어 모델인 TOE(TCP Offloading Engie) 제시하였다. 그리고 이 하드웨어에 대응하여 TCP 프로토콜을 지원할 수 있는 소프트웨어인 TOE manager를 설계하고 실제로 구현하였다. 그리고 이를 현재 가장 대표적인 웹서버 NGINX에 패치를 적용해서 동작을 확인하는 것으로 검증하였다. 마지막으로 TOE의 포텐셜을 확인하기 위해서, NGINX를 TOE 위에서 돌렸을 때와 Linux 커널의 TCP 스택을 사용했을 때의 시간 당 요청 처리 속도를 비교하였다. 그 결과 Linux의 TCP stack과 비교하여 동일 코어로 2.5~3배 높은 성능을 달성하거나, 같은 성능을 2~4배 적은 코어로 달성할 수 있었다.

제 2 장 TCP 스택 분석

제 1 절 TCP 프로토콜

TCP 프로토콜은 가장 대중적으로 사용되는 트랜스포트 계층(transport layer) 프로토콜이며, 멀티 네트워크를 사용하는 애플리케이션에게 안정적인 데이터 전송을 제공하기 위해 만들어졌다. 이를 위해 TCP는 애플리케이션에 데이터 송수신(data transfer), 안정성(reliability), 흐름 제어(flow-control), 멀티플렉싱(multiplexing), 커넥션 관리(connection management) 등을 기능을 제공한다.[11]

데이터 송수신: TCP 프로토콜은 데이터 송수신을 위해서 두 노드 쌍의 데이터 송수신 정보를 연속적인 데이터 스트림 형태로 저장한다. 각각의 데이터 스트림에는 데이터들의 시퀀스 넘버(sequence number)와 현재 상태가 적혀 있다. TCP는 이를 보고 몇 번째 데이터까지 전송/수신이 이루어졌는지 파악하고, 특정 데이터의 송수신 여부를 결정할 수 있다.

안정성: TCP 프로토콜은 데이터의 손상, 손실, 중복 그리고 비순차적인 전송이 일어나더라도 애플리케이션에 안정적인 데이터 송수신을 제공해야 한다. 이를 위해서 TCP 프로토콜은 각 데이터 패킷 수신 시 데이터가 가지고 있는 시퀀스 넘버로 ACK (acknowledgement) 시그널을 상대편으로 전송한다. 이를 통해서 상대편은 데이터의 전송이 이루어지는지 확인할 수 있고, 타임아웃 간격(timeout interval)동안 ACK이 도착하지 않았을 경우, 데이터를 다시 전송함으로써 안정적인 데이터 전송을 보장한다.

흐름 제어: TCP 프로토콜은 데이터 링크를 효율적으로 활용하기

위해서, 데이터 전송의 성공/실패 여부를 이용하여 인-플라이트-패킷(in-flight packet)의 수를 조절하는 알고리즘을 가지고 있다. 이를 통해 노드 페어는 네트워크 링크의 스피드를 최대한 활용할 수 있으며, 여러 노드 페어가 동시에 한 개의 네트워크 링크를 공유하더라도 공평하게 네트워크 링크의 처리량을 나누어 가질 수 있다.

멀티플렉싱: 한 개의 호스트 CPU에서 여러 TCP 연결을 동시에 사용하기 위해서, TCP는 각 호스트에게 주소(address)와 포트(port)를 식별자(identifier)로 제공한다. 호스트 CPU는 주소와 포트 쌍에 대응하는 소켓(socket)을 만든다. 그리고 연결이 이루어지면 각 호스트에는 (자신의 주소, 자신의 포트, 상대의 주소, 상대의 포트)의 식별자를 가진 소켓을 각자 가지게 된다. 그럼 각 노드는 자신의 소켓을 활용하여 반대편으로 데이터를 송수신 할 수 있다.

커넥션 관리: TCP 프로토콜을 제대로 수행하기 위해서는 호스트는 각 연결별로 송수신이 어디까지 이루어졌는가에 대한 정보가 필요하다. 이를 위해서 호스트는 CPU는 연결별로 송수신 데이터 스트림이나 소켓과 같은 정보를 TCB(TCP control block)에 담아 호스트 CPU에 저장하여 관리한다. 이를 위해서는 TCP 프로토콜은 두 노드가 통신을 원할 때 각 호스트에 연결에 대한 TCB를 생성하고 초기화해야 한다. 그리고 또한 데이터 송수신시의 TCB 정보를 읽거나 업데이트하면서 통신을 지원한다. 마지막으로 두 노드가 연결을 종료하였을 때 각 노드에서 TCB에 사용된 자원을 해방시켜 해줘야 한다.

제 2 절 TCP 프로토콜의 연결 상태 관리

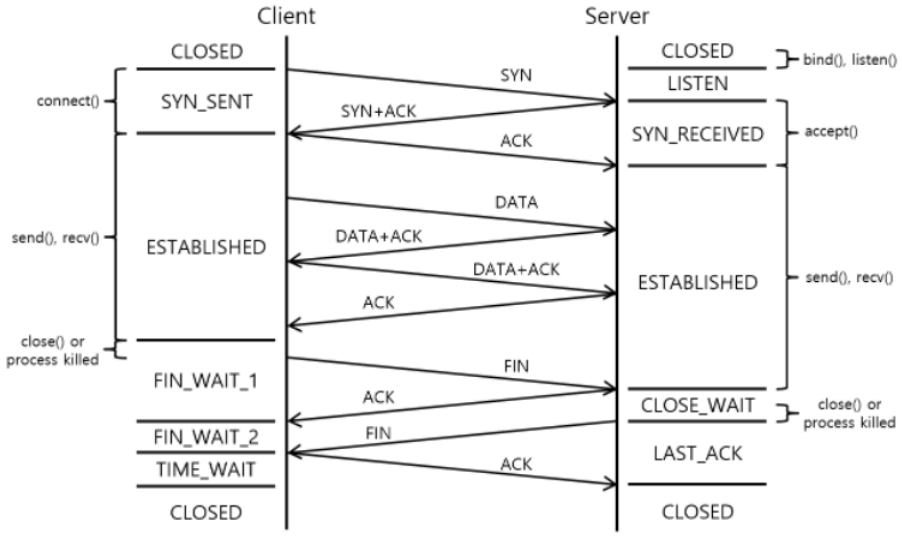

그림 2 TCP 연결 상태(connection state) [24]

그림 2는 TCP의 연결 상태의 변화를 시간의 흐름에 따라서 나타낸 것이다. 연결이 설립(established)되기까지의 과정을 풀어서 설명하면 다음과 같다.

0. 서버와 클라이언트의 두 소켓은 연결이 시작되기 전 CLOSED 된 상태로 시작한다.
1. 먼저 서버 쪽에서 연결 받고자 하는 소켓을 listen()함수를 통해서 CLOSED->LISTEN 상태로 바꾼다. LISTEN 상태의 소켓은 클라이언트로부터 SYN flag를 가진 패킷이 들어왔을 경우 새로운 연결을 생성할 수 있다.
2. 클라이언트에서 새로운 연결을 생성하기 위해서 connect()함수를 부른다. 클라이언트는 SYN flag를 가진 패킷을 서버의 LISTEN 상태의 소켓으로 전송한다. 그리고 이를 전송한 소켓의 상태를 CLOSED->SYN_SENT로

전환한다.

3. LISTEN 상태의 소켓에 SYN flag를 가진 패킷이 도착할 경우, Listen queue에 이를 저장해 둔다. 이후 accept() 시스템 콜이 호출될 경우, Listen queue에 저장된 패킷들을 보고 이를 위한 서버 소켓을 새롭게 생성한 후에 그 소켓에서 connect()를 요청한 클라이언트의 소켓으로 SYN+ACK flag를 올려서 전송한다. 그후 새롭게 생성한 소켓의 상태를 CLOSED->SYN_RECEVIED 상태로 변경한다.
4. 클라이언트에 SYN 패킷을 보낸 소켓에 SYN+ACK flag를 가진 패킷이 도착하면 connect()함수를 빠져나오고, 그 소켓의 상태를 SYN_SENT->ESTABLISHED로 전환하고 이를 ACK을 SYN+ACK을 보낸 서버 소켓으로 ACK flag를 올린 패킷을 전송하다.
5. 서버의 SYN_RECEIVED 소켓에서 ACK flag가 올라간 패킷을 받으면 소켓의 상태를 SYN_RECEVIED->ESTASBLISHED로 전환하고 연결의 설립을 완료한다.

1~5까지 서버, 클라이언트 간 SYN, SYN+ACK, ACK을 교환하는 과정을 3-웨이 핸드세이킹(3-way handshaking)이라고 한다. 이렇게 연결이 설립(established)된 소켓 쌍은 데이터를 자유롭게 주고받을 수 있다. 이후 연결을 종료 시에는 위와 비슷하게 4-웨이 핸드세이킹으로 FIN flag가 들어간 패킷을 주고받으면서 연결을 종료한다

제 3 절 TCP 스택의 호스트 CPU 사용

지금까지 네트워크 스택은 높은 유연성과 많은 연결을 지원할 수 있는 호스트 CPU 위에 구현되었다. 호스트 CPU의 강력한 프로세싱 파워는 네트워크 스택의 높은 퍼포먼스 달성을 가능케 하였고, 높은 프로그래밍 자유도는 TCP 스택이나 TCP 흐름 제어 알고리즘에 높은 구현 자유도를 제공하였다. 그리고 CPU의 잘 조직된 계층(hierarchy) 메모리 구조는 동시에 많은 연결이 있어서, 많은 TCB를 다루어야 하는 상황에서도 이를 큰 성능 하락 없이 지원해주었다.

하지만 네트워크의 고성능화가 이루어짐에 따라, 고성능의 네트워크에 네트워크 스택을 지원하기 위한 호스트 CPU의 사용량 또한 증가하기 시작하였다. 또한 [12,13]를 보면 데이터 센터에서 64~128 바이트의 작은 페이로드를 갖는 작은 패킷들을 처리해야하는 경우가 많은데, 작은 패킷의 경우 동일 데이터 처리하는데 더 많은 헤더를 분석해야 해서 네트워크 스택의 오버헤드가 커진다. 이러한 점 들로부터 가까운 미래에 TCP 스택의 높은 CPU 사용량은 치명적인 문제가 될 것이다. 이 문제의 심각성을 제대로 분석하기 위해서 프로파일링을 진행하였다.

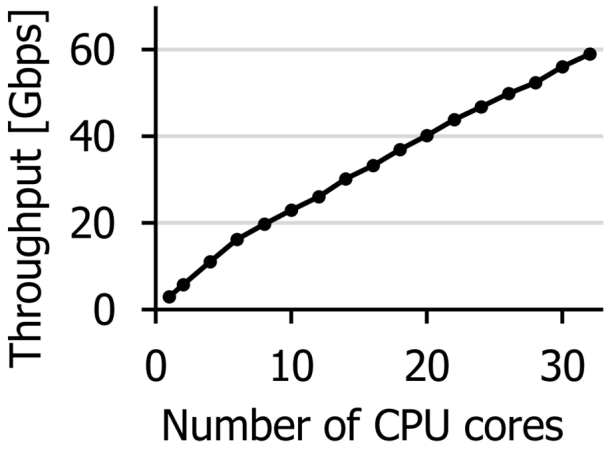

그림 3 iPerf의 처리량에 따른 호스트 CPU 사용량 (128byte 패킷)

그림 3은 iPerf[14]로 128바이트의 데이터에 대한 요청을 보낼 때, 요청을 받는 쪽의 호스트 CPU의 사용 코어 수를 바꾸어 가며 달성한 처리량을 나타낸 것이다. 이때 CPU의 코어별로 1개의 연결을 사용하였다. 이 실험에 Linux OS 기반의 Intel 5220 dual socket CPU가 사용되었으며, 두 노드는 100Gbps의 NIC으로 연결된 환경으로 세팅하였다. 또한 보수적인 결과를 얻기 위해 호스트 CPU 사용량을 줄여주는 TSO(TCP segment offload)[15]와 체크 섬 오프로딩(checksum offloading)[16] 와 같은 옵션을 킨 상태에서 진행하였다.

실험 결과 128바이트 크기의 페이로드에 대한 요청이 들어올 경우, 32개의 호스트 CPU 코어를 사용했음에도 60Gbps를 달성하였을 뿐 NIC의 100Gbps의 데이터 링크를 포화시키지 못했다. 만약 CPU 코어 수가 충분하다 하더라도, 800Gbps를 달성하는데 단순 계산으로 무려 430개의 호스트 CPU가 코어가 사용된다. 심지어 iPerf는 매우 간단한 요청을 보내는 코드로 구성되어, 받는 노드에서 처리 시 애플리케이션 코드의 수행을 거의

필요로 하지 않는다. 만약 웹서버와 같이 실제 워크로드로 테스트할 경우, TCP 네트워크 스택뿐만 아니라 http 헤더 해석 및 생성, 파일 시스템 접근과 같은 애플리케이션 코드의 수행이 필요하다. 이 경우 애플리케이션 코드로 인한 오버헤드의 더불어 콘텍스트 스위칭(context switching)이나 캐시 폴루션(cache pollution)과 같은 현상이 발생하여 코어당 네트워크 처리량을 감소시킬 것이다. 이러한 프로파일링 결과로부터 현재 Linux의 TCP 스택은 고처리량의 네트워크 링크를 낮은 패킷에서 처리하는 데 어려움을 겪는 것을 확인할 수 있었다.

제 4 절 선행 연구 분석

이처럼 TCP 스택을 사용하는데 지나치게 많은 호스트 CPU자원을 사용하는 문제는 여러 논문에서 지적되어왔다. 선행 연구들은 이에 대한 해결책으로 크게 3가지 방향의 해결책을 제시한다.

첫 번째는 TCP 네트워크 스택 자체를 경량화하는 것이다. [9,17,18,19의 논문들을 현재 Linux의 TCP의 네트워크 스택이 무겁고 비효율적으로 구성된 것을 원인으로 지적하였다. 먼저 MTCP[9]의 경우, 커널의 네트워크 스택을 사용하는 것이 아니라 유저 레벨의 가볍고 효율적인 네트워크 스택을 구현함으로써 이 문제를 해결할 수 있다고 주장하였다. IX, TAS[17,18]와 같은 경우에는 커널에 네트워크 스택을 재구성함으로써 이러한 문제를 해결할 수 있다고 주장하였다. 하지만 두 연구 모두 달성한 성능 향상이 수 배 정도로 제한적이었고, 달성한 성능이 10Gbps에 미치지 못했다. 또한 프로세스 간의 공유자원 접근 시 lock을 사용하여 locking 오버헤드(overhead), 또한 애플리케이션 코드가 네트워크 스택 코드의 공유로 일어나는 캐쉬 폴루션(cache pollution), 여러 코어 간의 캐시 코히런스(cache coherence) 문제등으로 인해 성능이 저하될 수 있다. 무엇보다 CPU core수가 제한되어 있는 만큼 TCP 프로세싱을 위한 연산 유닛을 늘릴 수 없어 한계가 명확하다는 단점 또한 가지고 있다. 이러한 이유로 단순히 네트워크 스택을 경량화 하는 것은 미래의 수 백 Gbps 네트워크에 TCP 프로토콜을 지원하기에 적합한 솔루션은 될 수 없다고 판단하였다.

두 번째는 스마트 닉과 같은 디바이스 CPU에서 TCP 네트워크 스택을 처리하는 데 사용하는 방법이다. 디바이스 CPU의

경우, 호스트 CPU의 사용을 절약하면서, 호스트 CPU의 장점인 유연성을 가져갈 수 있다는 장점을 가지고 있다. 하지만 디바이스 CPU의 경우 호스트 CPU에 비해서 훨씬 낮은 프로세싱 파워를 가지며, 이로 인해서 지원할 수 있는 TCP의 성능이 크게 제한된다. [20] AccelTCP[21]처럼 디바이스 코어에 TCP 스택을 부분적으로 오프로딩하여 보조하도록 하는 연구도 있다. 하지만 이 경우 TCP 스택을 디바이스 코어와 CPU 코어 간의 TCB이동으로 인해서 오프로딩할 수 있는 부분이 연결 생성, 종료 등으로 극히 제한되며, 이로 인한 성능 향상도 제한(marginal)적이다. 이러한 문제로 인해서 디바이스 CPU 또한 수백 Gbps의 미래의 네트워크를 지원하는 데는 적합하지 않다고 판단하였다.

마지막은 TCP 스택을 위해 하드웨어를 새롭게 설계하고 오프로딩하여 활용하는 것이다. 이 방향의 선행 연구 Limago[23], Tonic[23]들은 다른 방향의 연구에 비해서 고성능의 네트워크에 TCP 프로토콜을 지원하였다. 특히 Tonic[23]의 경우 조건부이지만 100Gbps에 TCP 프로토콜을 달성하는 성과를 이루었다. 이는 하드웨어 로직을 CPU와는 다르게 TCP 네트워크 스택에 특화시킬 수 있기 때문이다. 또한 필요에 따라 하드웨어 로직 양을 늘리거나, 파이프라인을 구성함으로써 높은 성능을 달성할 수 있다. 특히 TCP 스택의 경우 각 연결들이 완전히 독립적이므로, 하드웨어의 각기 다른 연산 유닛과 메모리 모듈에서 병렬적으로 처리하기에 적합하다. 이러한 사실들로 미루어 볼 때, 미래의 수백 Gbps에 네트워크에서 TCP 스택을 사용 시 하드웨어로 오프로딩하는 것은 가장 유망한 솔루션으로 판단하였다.

제 3 장 TCP 오프로딩 하드웨어 모델

제 1 절 TONIC의 TCP 오프로딩 하드웨어

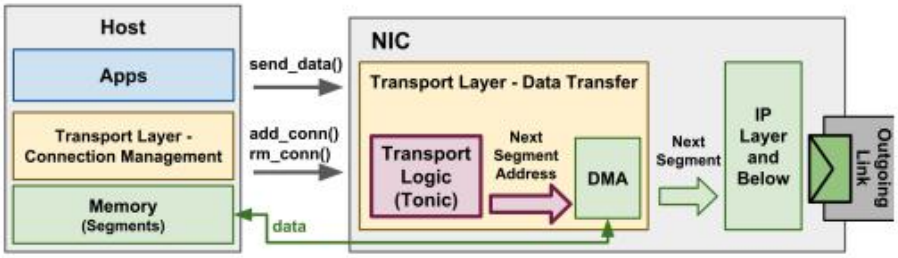

그림 4 TONIC의 하드웨어 오버뷰 (sender side) [23]

TCP 하드웨어 오프로딩 모델을 설계하기 위하여, 현재 TCP 오프로딩 선행 연구 중 가장 높은 성능을 달성한 TONIC[23]의 구조 분석하였다. 그림 4는 TONIC의 수신(sender) 쪽의 아키텍처를 나타낸 것이다. 먼저 TCP 스택 중에 연결 관리 코드를 부분을 제외하고 대부분의 TCP 스택이 NIC에 오프로딩 되어있다. 또한 호스트 메모리와 NIC의 메모리가 DMA를 통해서 데이터를 주고받는 방식이다. NIC의 TCP 수행 로직은 호스트로부터 커맨드를 받아서 이에 대한 요청을 처리하는 방식으로 동작한다.

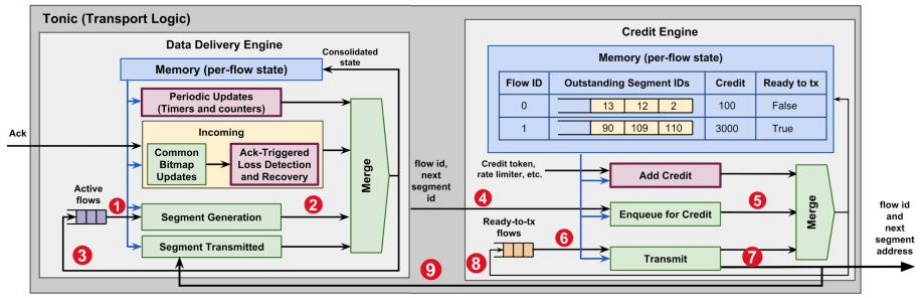

그림 5 TONIC의 아키텍처 [23]

TONIC이 TCP 프로토콜을 소프트웨어가 어떻게 지원하는지 설명하도록 하겠다. 먼저 연결 관리 부분을 살펴보면, 연결의 설립(setup)과 종료(tear down)의 경우 TONIC이 아닌 호스트 CPU

커널에서 일어난다. 호스트 CPU의 연결을 설립한 이후, TONIC으로 주소와 포트와 같은 멀티플렉싱을 위한 식별(identifier) 정보, 연결 상태 정보를 넘겨준다. TONIC은 이 연결에 대해서 TCB를 생성하고 플로우 아이디(flow id)를 부여하고 이를 호스트 CPU에게 알려준다. 호스트 CPU나 TONIC은 이 연결에 대한 정보를 플로우 아이디를 식별하여 통신하게 된다.

데이터 송신 부분을 살펴보면, 소프트웨어는 연결에 주어진 송신 버퍼의 데이터를 삽입하고, 데이터 전송에 사용할 세그먼트의 크기를 알려주는 것으로 소프트웨어의 역할은 끝난다. TONIC은 먼저 보낼 데이터를 세그먼트 단위로 쪼갠 후에 보낼 세그먼트를 기록한 메모리(그림 5의 오른쪽 파란색 테이블)에 삽입한다. 이후 TONIC이 연결의 흐름 제어 정보를 읽어서, 세그먼트들의 송신 가능 여부를 확인한 후 송신을 결정한다. 즉, 호스트 CPU가 데이터 송신 과정에서 하드웨어에 커맨드를 내려주는 것과 데이터를 넘겨주면 TONIC 데이터 송수신, 안정성, 멀티플렉싱과 흐름 제어를 하드웨어에서 자동으로 관리해주는 방식이다.

그래서 본 연구에서는 TONIC의 방식을 채용하여, 하드웨어 모델에서는 호스트 CPU가 하드웨어에 커맨드를 내려주는 것과 데이터를 복사해주거나, 해오는 것만으로 데이터 송수신을 사용할 수 있는 하드웨어 모델을 설계할 것이다.

제 2 절 기존 TONIC 하드웨어의 한계점

TONIC은 하드웨어 오프로딩을 통해서 100Gbps 스케일의 고속 데이터 링크에 TCP 프로토콜을 지원할 수 있는 하드웨어 오프로딩의 잠재력을 보여주었지만, 4 가지 한계점을 가지고 있다.

첫 번째로 TONIC은 연결 관리를 호스트의 커널에 의존한다. 이렇게 네트워크 스택에 호스트 커널이 들어가게 되면, [9,17,18]에서 지적했듯이 호스트 CPU에서 애플리케이션과 커널 코드 사이의 콘텍스트 스위칭과 캐쉬 폴루션과 같은 문제가 발생하게 된다. 또한 패킷의 처리 위치가 일원화되지 않은 것도 큰 문제이다. 각 패킷이 현재 TCP 연결/종료 단계인지 데이터 송수신 단계인지에 따라 패킷의 처리 위치가 달라진다. 그러므로 NIC에서 하드웨어로의 패킷 포워딩(forwarding)이나 프리패칭(prefetching)을 하기 어렵다는 문제점이 있다. 또한 커널에서 연결을 설립하고 넘겨주는 방식은 연결 후 첫 데이터 송수신까지의 시간이 오래 걸린다는 단점이 있다.

실제 TCP 동작의 경우 3-웨이 핸드세이킹(3-way handshaking)에 마지막 ACK 패킷에 데이터가 같이 보내지는 경우가 많다. TONIC의 경우 데이터 처리가 호스트에서 연결을 완료한 후에 TONIC으로 넘겨주고, 하드웨어에 TCB 블록 할당이 이루어진 후에야 데이터 전송이 이루어지므로, 패킷의 ACK과 같이 보내진 첫 데이터의 처리에는 이러한 시간이 포함되어 상당히 긴 레이턴시가 발생한다. 이는 적은 수의 요청을 보내고 연결을 종료하는 단순 리퀘스트 요청이 많을경우 심각해질 것이다.

두 번째로 TONIC은 부족한 프로세싱 파워로 인해서 TCP의 바이트 스트림(Byte stream)을 포기하고, 데이터의 처리 최소 단위를 세그먼트로 고정시켰다. 이로인해서 패킷의 실제 크기와 상관없이

세그먼트의 배수로만 데이터를 보낼 수 있다. 만약 세그먼트가 패킷의 페이로드에 비해서 너무 크면 내부 단편화(Internal Fragmentation) 문제가 발생하며, 세그먼트가 페이로드에 비해서 너무 작을 경우 헤더로 인한 오버헤드가 커지게 된다. 또한 TSO와 같은 추가 최적화를 구현하기 어렵다.

세 번째로 데이터의 수신과 관련하여 설명이 부족하고, 엔드-투-엔드로 구현된 풀 스택을 보여주지 못한다는 것이다. TONIC은 데이터 송신에 대해서만 다루고 있으며, 수신에 대해서 어떻게 다루는지 설명하고 있지 않다. 또한 소프트웨어와 하드웨어가 어떤 방식으로 통신하여 데이터 송수신을 요청하고 TCP 상태 정보를 싱크하는지에 대한 설명이 부족하다. 하지만 TONIC의 경우 이를 어떻게 처리하고 있는지에 대한 설명이 없고, 엔드-투-엔드로 워크로드를 돌린 것을 보여주지 않아서 이런 부분에 대한 구현이 제대로 되어 있는지 알기 어렵다. 실제로 100Gbps를 달성하는 상황에서 소프트웨어에 얼마나 오버헤드를 발생시키는지에 대한 논의가 없다.

마지막으로 연결수가 늘어난 경우에 대한 논의가 없었다. 기존의 하드웨어의 SRAM에서 지원하는 것보다 더 많은 연결이 시도될 경우, TCB의 공간 부족으로 새로운 연결을 지원할 수 없다. 이를 방지하기 위해서는 TCB를 저장하기 위한 공간이 부족할 경우 오랫동안 사용되지 않는 연결의 TCB를 DRAM과 같은 하위 메모리로 내보내고(evict) 현재 사용하는 연결을 올려주는(promotion)시킬 수 있는 메모리 서브시스템을 갖추어야 한다.

본 연구에서는 이러한 한계점을 극복할 수 있는 TCP 오프로딩 하드웨어를 새롭게 제시하고자 한다.

제 3 절 TOE 디자인 목표

제1절, 2절에서는 선행연구인 TONIC의 TCP 오프로딩 하드웨어가 소프트웨어와 어떻게 협력하여 TCP 프로토콜을 지원하는 방법과 TONIC의 한계점을 살펴보았다. 본 절에서는 이 내용을 하여 미래의 수백 Gbps를 위한 하드웨어의 모델에 대해 다음과 같이 4가지 목표를 제시한다.

1. 호스트 CPU의 부담을 최소화하기 위해서 TCP 스택을 자체적으로 처리할 수 있어야 한다.

2. TCP의 바이트 스트림과 같이 TCP가 기존의 제공하는 기능을 유지할 수 있어야한다.

3. 데이터 수신을 포함하여 실제 워크로드를 엔드-투-엔드로 지원할 수 있어야 한다.

4. 지원하는 연결 수보다 많은 연결이 들어올 경우에 대비하여 필요에 따라 TCB를 외부 메모리로 보내거나, 외부 메모리로부터 TCB를 내부 메모리로 가져올 수 있어야 한다.

각각의 디자인 목표를 풀어서 설명하면 다음과 같다. 첫 번째로 TOE는 호스트의 도움 없이 자체적으로 TCP 프로토콜을 지원할 수 있어야 한다. TOE의 경우 호스트로부터 현재 LISTEN 상태의 소켓의 주소와 포트와 같은 식별 정보를 받아 두었다가, 상대 쪽에서 그 주소와 포트로 연결 요청(connect) 패킷이 올 경우, 혼자서 그 연결에 대한 설립을 완료시킬 수 있어야 한다. 즉 호스트를 거치지 않고 TCB를 할당한 후 직접 3-웨이 핸드세이킹 완료할 수 있어야 한다. 데이터 송수신도 호스트와 독립적으로 처리해야 하며 호스트로 처리 결과를 알려주는 방식으로 동작 해야 한다. 연결 종료도 호스트 쪽에서 종료를 요청하든지(active close), 상대편 쪽에서 종료

를 요청했든지 (passive close)에 상관없이 하드웨어에서 처리를 완료한 후에 호스트에 알리는 방식으로 동작해야 한다. 이러한 동작 방식은 호스트의 응답을 기다리지 않아서 패킷 처리 속도를 높이고, 호스트 CPU의 사용량도 줄일 수 있다. TCP 프로토콜을 사용할 수 있다.

두번째로 원래 TCP 기능을 지원하기에 하드웨어가 충분한 프로세싱 파워들을 가져야한다. 이때 필요한 프로세싱 파워를 얻기 위해서, TCP를 처리하기 위한 처리하는 코어를 여러 개 두고 각각의 다른 연결들을 분배함으로써 많은 코어에서 독립적으로 TCP 프로세싱을 처리할 수 있도록 한다.

세번째로, 엔드-투-엔드로 워크로드를 수행할 수 있어야한다. 즉 애플리케이션에서 기존 TCP에 대응하는 동작을 요구하였을 때 TOE는 애플리케이션에 Linux를 실행했을 때와 같은 결과 돌려주어야 한다.

마지막으로 연결들의 TCB를 저장하는 공간이 하드웨어 내부 메모리(SRAM)보다 많더라도 연결 정보를 읽지 않고 오류 없이 동작해야 한다. 이를 위해서 사용 빈도가 높은 연결(hot connection)을 외무 메모리에 위치시키고, 사용 빈도가 낮은 연결들(cold connection)들은 내부 메모리가 부족할 경우, 외부 메모리(DRMA)으로 위치시켰다가, 필요할 때 내부 메모리로 올릴 수 있어야 한다.

제 4 절 TOE 아키텍처

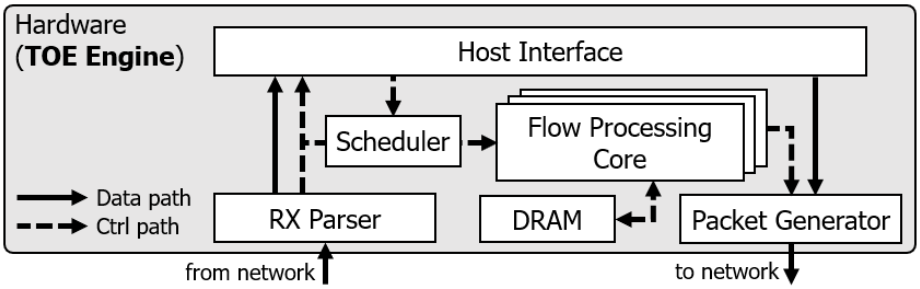

그림 6 TOE(TCP offloading engine) 모델

그림 6은 앞에서 이야기한 가이드 라인을 바탕으로 설계한 TOE의 아키텍처를 모듈 별로 나타낸 것이다. 각 모듈은 다음과 같은 역할을 한다.

RX Parser: 외부로부터 들어온 패킷을 해석하는 부분이다. 외부로부터 패킷이 들어오면 패킷의 식별정보(출발 주소, 도착 주소, 출발 포트, 도착 포트)를 분석하여, 이 패킷이 어느 연결에 대한 패킷인지 식별해준다. 이 과정을 통해 패킷의 종류(SYN, ACK, FIN data send, data recv)와 패킷이 현재 TOE에서 몇 번째 세션(session)으로 관리되고 있는지를 판명한다.

Scheduler: RX Parser를 거친 패킷은 어떠한 세션에 속해 있는지 파악할 수 있다. Scheduler는 이 패킷을 현재 그 세션을 처리하고 있는 FPC(Flow processing core)로 전달해주는 역할을 한다. 만약 모든 FPC들에 추가로 세션을 처리할 수 있는 여력이 없을 경우, 일부 세션을 외부 메모리로 내보내어 공간을 확보한다.

Flow Processing Core: FPC는 패킷이 도착하거나, 호스트로부터 요청이 왔을 때 TCP 프로세싱을 담당하는 코어이다. FPC 내부의 메모리에는 자신이 담당하는 세션들의 TCB들을 저장해두고, 흐름 제어와 데이터 송수신을 보장한다. 이러한 과정이 끝나면 Packet

generator에게 상대편으로 패킷을 보낼 것을 요청한다.

Packet generator: FPC로부터 요청을 받아, 상대편의 보낼 패킷을 생성하고 송신한다. 이 과정에서 필요에 따라 SYN, ACK, FIN 등의 flag를 올릴 수 있다.

Host interface: 호스트와 통신하는 부분이다. 컨트롤 패스(path) 관리와 데이터 패스 관리를 역할로 나눌 수 있다. 컨트롤 패스 관리의 경우, TOE에서 연결 설립(connection established), 데이터 수신, 데이터 송신 완료, FIN 요청과 같이 호스트에 알려야 할 이벤트가 발생하였을 때 이를 호스트에 커맨드를 보내서 알려주는 활을 한다. 또한 호스트로부터 TOE에 새로운 Listen 소켓 생성, 데이터 송신등 하드웨어에게 알려야 할 때, 호스트로부터의 커맨드를 읽어서 TOE에 알려주는 역할을 한다. 데이터 패스의 경우 수신 패킷의 데이터를 적절한 호스트 코어의 DMA 영역에 복사해두거나, 송신 패킷의 데이터를 호스트로부터 읽어와서 TOE에 DMA 영역에 복사해둔다.

제 4 장 TOE Manager 설계 및 구현

본 장에서는 제3절에서 이야기한 TOE(TCP offload engine)와 애플리케이션 사이를 중계해주는 소프트웨어인 TOE Manager를 어떻게 구현했는지 설명하겠다.

제 1 절 TOE와의 인터페이스

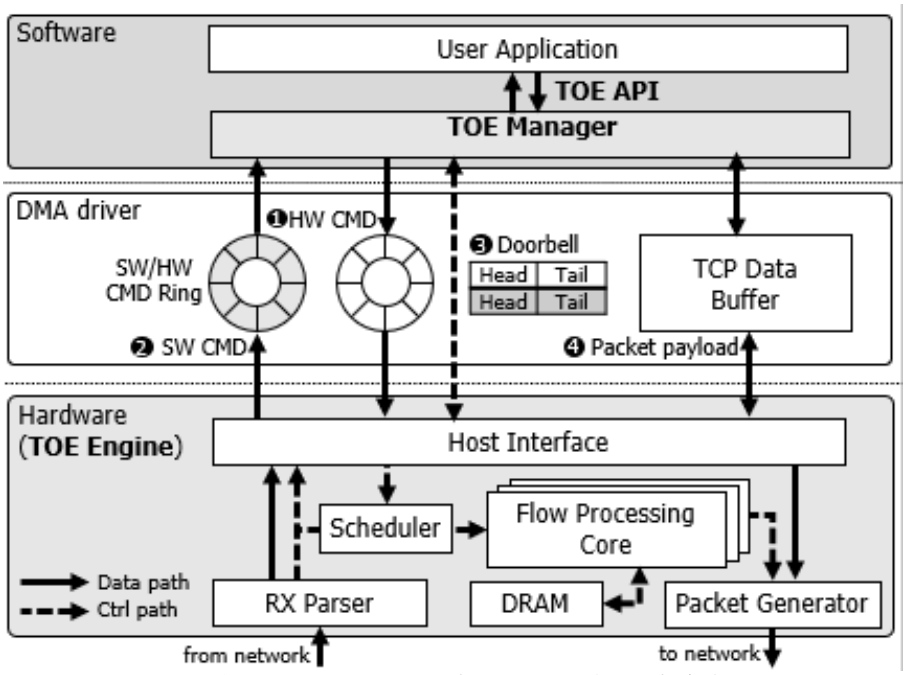

그림 7 TOE Manager와 TOE 간의 인터페이스

그림 7은 TOE Manager와 TOE 간의 인터페이스를 설명한 것이다. TOE와 TOE Manager는 DMA 영역을 통해서 데이터나 커맨드를 주고받는다. TOE Manager는 초기화 단계에서 DMA 영역에 SW와 HW의 커맨드를 저장할 SW/HW CMD Ring 과 TCP 패킷의 데이터를 저장할 수 있는 TCP Data Buffer를 생성하고, TOE의 알려준다. 각각의 CMD는 커맨드 타입과 플로우 아이디 그리고 추가 파라미터가 128비트로 패킹(packing)되어 있다. 또한 TOE manager 또는 TOE 커맨드는 각자의 CMD Ring에 채웠을 때

MMIO를 통해서 이를 알려준다.

각각의 인터페이스를 자세히 설명하면 다음과 같다.

SW CMD: TOE Manager에서 TOE로 보내는 커맨드를 의미한다. 커맨드의 종류와 그에 따른 동작은 다음과 같다.

- Listen: 애플리케이션인 특정 주소와 포트의 소켓을 LISTEN 상태로 변경시에 이를 TOE에 알려주는 커맨드다. TOE Manager에 그 소켓의 주소와 포트를 알려준다. 이후 TOE는 이 주소와 포트로 연결 요청이 오는 것을 대기하게 된다.
- Connect: 애플리케이션인 특정 주소와 포트의 소켓을 연결을 요청 시 그 요청을 TOE에 알려주는 커맨드다. TOE Manager에 그 소켓의 주소와 포트를 알려준다. 이후 TOE는 이 주소와 포트로 SYN 패킷을 보낸다.
- Send: 애플리케이션이 특정 연결로 데이터를 보낼 때, 보내고 싶은 데이터를 그 연결에 대응하는 TCP Data Buffer에 저장한 후에 TOE예 데이터를 보낼 것을 요청하는 시그널이다. TOE Manager는 보내고 싶은 연결의 플로우 아이디와 보내고 싶은 데이터의 크기를 같이 보내준다. 그럼 TOE는 이 플로우 아이디에 대응하는 TCP data buffer로부터 데이터를 읽어 가서 상대편에게 보내준다.
- Recv: 애플리케이션이 특정 연결의 데이터를 읽기 요청을 보냈을 때, TCP data buffer로부터 데이터를 읽은 후에 데이터 읽기가 완료되었음을 알리는 커맨드다. TOE Manager 플로우 아이디와 읽은 데이터의 크기를 같이 보내준다. 그럼

TOE는 TCP Data buffer가 비워졌음을 확인하고, 이후 같은 연결에 새로운 패킷이 들어왔을 때, 기존 데이터 영역에 덮어쓰는 것을 허용한다.

- Close: 애플리케이션이 특정 연결의 종료 요청을 보냈을 때, TOE Manager에서 이를 TOE에 알리는 커맨드다. TOE가 커맨드를 받았을 때, 혼자서 4-웨이 핸드세이킹을 통해 연결을 종료한다. 그러므로 TOE Manager는 이 커맨드를 보내면서 그 연결에 대한 자원을 모두 반납한다.

HW CMD: TOE가 TOE Manager로 보내는 커맨드를 의미한다. 커맨드 종류에 따른 동작은 다음과 같다.

- Listen established: TOE가 TOE manager에게 listen 소켓의 connect 요청이 와서 연결이 생성되었음을 알리는 커맨드다. TOE는 어떤 listen 소켓에 대한 요청인지와 새로운 연결의 플로우 아이디를 알려준다. TOE Manager는 새로운 연결을 위한 TOE 소켓을 준비하고, 애플리케이션에서 accept() 요청이 왔을 때 처리하기 위해, listen 소켓의 리스너 큐(Listener queue)에 삽입해 둔다.
- Connect established: TOE가 TOE manager가 요청되었던 connect에 대한 연결이 생성되었음을 알리는 커맨드다. TOE는 새로운 연결의 플로우 아이디를 알려준다. TOE Manager는 connect 대기 상태에서 빠져나와서 연결이 완료되었음을 애플리케이션에게 알린다.
- Data received: TOE가 TOE managers에게 특정 연결에 데이터가 도착했음을 알리는 커맨드다. TOE는 데이터가 도착한 연결의 플로우 아이디와 도착한 데이터의 크기를 알려

준다. TOE Manager는 그 소켓의 읽을 데이터가 있음을 기록해 두어, 이후 애플리케이션이 데이터를 요청이 왔을 때 돌려줄 수 있도록 한다.

- Send complete: TOE가 TOE manager가 요청했던 데이터 송신이 완료되었음을 알리는 커맨드다. TOE는 데이터를 보낸 연결의 플로우 아이디와 그 크기를 알려준다. TOE Manager는 데이터의 전송이 완료되었음을 알아차리고, 이후의 애플리케이션으로부터 send 요청이 오면 기존의 데이터 영역에 덮어씌우는 것을 허용한다.
- Close: TOE가 TOE managers에게 상대편으로 CLOSE 요청이 도달했음을 알리는 커맨드다. TOE는 데이터를 보낸 연결 종료가 이루어진 플로우 아이디를 알려준다. TOE가 혼자서 4-웨이 핸드세이킹을 통해 연결을 종료할 것을 믿고, TOE Manager는 그 연결에 대한 자원을 모두 반납한다.

Doorbell: TOE Manager와 TOE가 자신의 커맨드를 보냈을 때, MMIO를 통해 CMD Ring의 Doorbell을 업데이트 함으로써 상대에게 알려준다. CMD를 보낸 쪽은 보낸 CMD수 만큼 헤드 포인터(head pointer)를 전진시키고, CMD를 받은 쪽은 처리한 CMD수 만큼 테일 포인터(tail pointer)를 전진시킨다.

TCP data buffer: TCP 패킷의 들어갈 정보를 저장하는 buffer에는 buffer에는 플로우 아이디를 인덱스로 하는 송신용 버퍼와 수신용 버퍼가 각각 64KB씩 존재한다.

제 2 절 TOE Manager의 구현

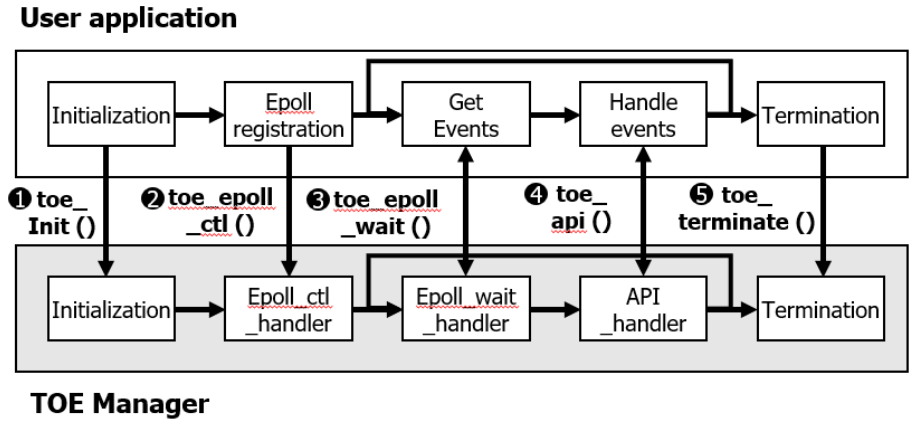

그림 8 TOE Manager의 실제 동작

그림 8은 Epoll을 사용하는 TCP 애플리케이션에 TOE Manager가 TCP 프로토콜을 제공하는 것을 나타낸 것이다. 애플리케이션이 TOE를 사용하기 위해서 기존의 코드를 거의 수정할 필요가 없다. TOE manager가 등록되어 있는 라이브러리를 추가하고, TCP API들에 접미사 toe_ 를 붙이기만 하면 TOE manager가 TCP가 TOE에서 수행되도록 이어주기 때문이다.

일반적으로 Epoll을 포함한 TCP 애플리케이션이 TOE manager를 사용할 경우 실제는 다음과 같이 동작한다.

1. 애플리케이션이 TOE를 사용하기 위해서 먼저 toe_init()함수를 호출하여 TOE Manager를 초기화해준다. 이 때 TOE Manger는 TOE Manager를 수행하는 필요한 memory space를 잡고 초기화하며, TOE와의 DMA 연결을 설정한다.
2. Epoll을 사용하는 애플리케이션은 epoll_ctl()함수를 통해 소켓 별로 관심있는 이벤트들을 등록해야 한다. 애플리케이션에서 toe_epoll_ctl() 함수를 호출하면 TOE Manager는 그 소켓의 관심 이벤트를 추가/변경/삭제한다. 이렇게 관심 이벤트

에 등록되면 나중에 TOE로부터 이와 관련된 HW CMD가 도착했을 때 레디 리스트(ready list)에 이벤트를 등록해 두었다가 d이후 애플리케이션이 toe_epoll_wait() 함수를 호출 시 레디 리스트에 있는 이벤트들을 돌려준다.

3. 애플리케이션은 관심 있는 이벤트를 등록해둔 후, 그 이벤트도 도달하기를 바라면서 toe_epoll_wait()함수를 호출한다. 그러면 TOE Manager는 먼저 도착했던 HW 커맨드를 읽고 TCP 상태를 업데이트한다. 업데이트 도중에 관심 있는 이벤트가 발생했을 경우 준비 리스트(ready list)에 추가한다. HW CMD의 읽기 과정이 끝나면. 준비 리스트를 점검해서 이벤트들이 여전히 유효한지 체크하고, 유효한 이벤트들을 애플리케이션으로 돌려준다. 만약 유효한 이벤트가 없다면 polling하면서 TOE로부터 새로운 HW CMD가 도착하기를 기다린다.
4. 애플리케이션은 이벤트를 돌려받은 후 이에 대응하는 동작을 TCP API를 통해서 요청한다. 몇 가지 예시를 들자면, 연결 설립 이벤트를 받았을 경우 toe_accept()를 호출하여 연결을 받아들인다. 또한 데이터 수신 이벤트를 받았을 경우 toe_recv()를 호출하여 애플리케이션으로 데이터를 읽어온다. 송신할 데이터가 있는데, 현재 TCP 송신 버퍼가 비어 있다면 toe_send()를 통해 데이터 송신을 요청한다. 이렇게 3~4를 반복하면서 애플리케이션은 TCP 프로토콜을 사용한다.
5. 마지막으로 애플리케이션 코드가 종료되기 이전에 toe_terminate()함수를 호출하여, TOE Manager가 정리를 할 수 있도록 해준다. 이때 TOE Manager는 사용한 자원을 반납하고, TOE와의 DMA 연결을 해제한다.

제 3 절 멀티 코어로의 확장

본 장 제 1, 2절에서는 TOE Manager가 어떻게 구현되었는지 설명하였다. 하지만 이 구현 방식은 호스트의 싱글 코어를 사용할 경우 잘 동작하지만, 위에서 설명한 소프트웨어 구조로는 멀티 코어로 확장시에 2 가지 문제점이 있다. 첫 번째로 TOE manager는 통신이 없어서 idle한 시간에 HW CMD를 polling하면서 대기하고 있기 때문에, 항상 CPU를 100퍼센트 사용한다. 두 번째로 TOE가 HW CMD를 어떤 코어로 보내야할 지 알 수 없다는 것이다. 왜냐하면, TOE입장에서는 어느 코어에 위치한 TOE Manager가 이 연결을 어느 코어에 있는 처리 혹은 처리 예정인지 알 수 없기 때문이다. 이러한 문제는 TOE에서 각 core로 HW CMD를 전송시에 Interrupt 시그널을 보내게 하고, TOE가 연결별로 현재 처리하는 코어를 트랙킹(tracking) 함으로써 해결할 수 있다. 하지만 우선은 이를 담당하는 소프트웨어인 TOE Polling Manager 두고 이를 보조 코어에서 실행하는 것으로 이러한 문제를 해결하였다.

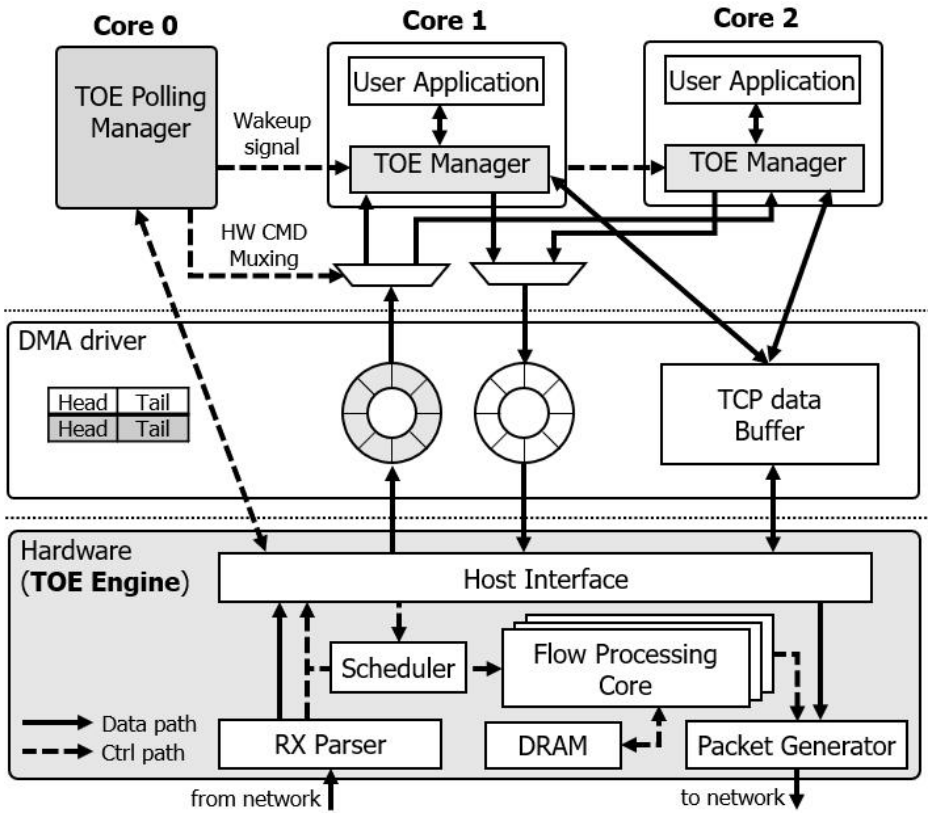

그림 9 Multi-core에서의 TOE Manager

그림 9는 멀티 코어에서의 TOE Manager를 나타낸 것이다. TOE Polling Manager는 HW와 통신하면서 HW CMD를 TOE Manager를 실행하는 다른 코어들에 분배하는 역할을 수행한다. 또한 TOE manager를 수행하는 코어에 TOE로부터 CMD가 오지 않아서 잠들었을 때, 인터럽트 시그널을 보내서 깨워주는 역할을 한다.

제 5 장 평가

제 1 절 평가 환경

TOE Manager를 엔드-투-엔드로 현재 가장 대중적으로 사용되는 NGINX[25] 웹서버를 사용하였다. NGINX 웹서버를 Linux의 TCP 스택을 활용하는 것과 TOE에 TCP를 오프로딩한 것을 비교하였다. 이를 위해서 NGINX에 32줄의 코드를 수정하여 TCP를 TOE를 통해 돌도록 패치 하였다. 또한 NGINX의 성능을 테스트하기 위해서 대중적인 벤치마크 툴인 wrk[26]를 활용하여, 같은 시간 동안 256바이트의 데이터를 요구하는 HTTP GET request에 대한 올바른 응답 횟수로 성능을 비교하였다. 이 때 Linux kernel은 5.3 버전을 사용하였다. 그리고 공평한 비교를 위해서 Linux와 TOE manager 모두 dual-socket 12-core Intel Xeon Gold 5118 CPU를 2.3 GHz를 사용하였다. 그리고 wrk와 NGINX를 실행하는 두 노드는 100Gbps를 지원하는 ConnectX-5로 연결한 상태로 실험을 진행하였다.

제 2 절 실험 결과

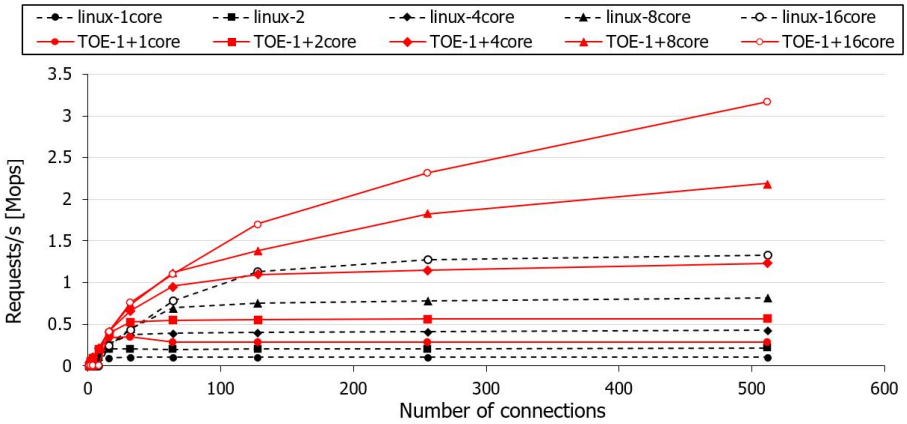

그림 10 Linux와 TOE에서 NGINX RPS 처리량

그림 10은 NGINX를 Linux의 TCP 네트워크 스택을 사용해서 돌렸을 때와 초당 request 처리 수를 나타낸 것이다. 주의할 점은 TOE의 경우, TOE Manager를 돌리는 코어 이외에 TOE Polling Manager를 다른 코어가 돌아가고 있다는 것이다. 실험 결과는 다음과 같다. Polling Toe Manager를 core를 제외하고 보았을 때, TOE는 같은 core수에서 2.5~3배의 성능향상을 보여주었다. 이는 코어를 2~4배를 더 써야 얻을 수 있는 성능이다. 이는 TOE를 사용할 경우, 기존의 성능대비 2.5~3배의 높은 성능을 달성하거나, 2~4배 적은 코어 수를 사용해서 동일한 성능을 낼 수 있다는 고무적인 결과를 의미한다. 특히 코어 수가 1에서 16까지 늘어나는 동안에도 이 경향성이 유지되므로, 코어 수에 비례해서 절약하는 코어 수도 늘어날 것으로 예측할 수 있다.

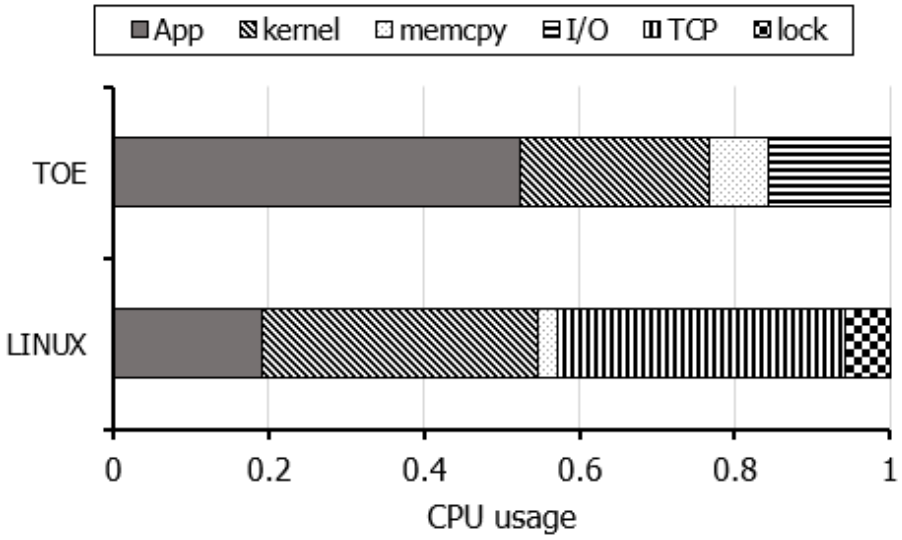

그림 11 Linux 와 TOE에서 CPU breakdown

그림 11은 NGINX를 Linux와 TOE에서 돌렸을 때의 CPU breakdown을 나타낸 것이다. TOE는 Linux의 40%의 시간을 차지하던 TCP stack을 성공적으로 제거한 것을 확인할 수 있다. 이 덕분에 애플리케이션에게 2.8배의 CPU의 usage 시간을 제공하였고, 이는 그림 10에서 달성한 성능 향상과 일치하는 수치이다. TOE의 경우 네트워크 스택에서 kernel을 사용하지 않음에도 40% 정도의 시간을 kernel에 사용하는 것을 볼 수 있는데, 이는 vfs_read와 같이 request가 온 요청한 html 파일에 접근하는 과정에서 발생한 것이다.

제 6 장 결론 및 제언

본 연구에서는 미래의 네트워크의 고성능화가 이루어져서 수백 Gbps 수준에 도달하게 이에 TCP 프로로콜을 지원하기 위해서 호스트 CPU 사용량이 심각하게 많아진다는 점을 관찰하였다. 그리고 이에 대해서 선행 연구 분석을 통해, 기존 해결책들을 검토하였고 그중 TCP를 위해서 설계된 하드웨어에 TCP를 오프로딩하는 것이 적합하다고 판단하였다. 이를 위해서 오프로딩하는 하드웨어가 갖추어야할 조건들을 검토하고 이를 바탕으로 TOE를 설계하였다. 또한 TOE가 엔드-투-엔드로 TCP 스택을 지원할 수 있도록, 이를 위한 소프트웨어인 TOE Manger를 설계하였고 구현하였다. 그리고 실제 상용 웹서버인 NGINX로 테스트하여 TOE로 TCP 오프로딩시 2.5~3배의 성능 향상을 달성할 수 있음을 보여주었다. 앞으로는 실제 수 백 Gbps의 네트워크에 TCP 프로토콜을 지원할 수 있도록 TOE와 TOE Manager를 개선할 예정이다.

참고 문헌

[1] "Building Express Backbone: Facebook’s new long-haul network",[Online].Available:https://engineering.fb.com/2017/05/01/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/

[2] Kalin Ovtcharov, Olatunji Ruwase, et al.Toward accelerating deep learning at scale using specialized hardware in the datacenter. In 2015 IEEE Hot Chips 27 Symposium (HCS), pages 1-38. IEEE Computer Society, 2015.

[3] Yu Gan, Yanqi Zhang, Dailun Cheng, et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. page 3-18, New York, NY, USA, 2019. *Association for Computing Machinery*.

[4] Nikita Lazarev, Shaojie Xiang, et al. Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. *In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 36-51, 2021.

[5] NVIDIA DGX-2. [Online]. Available:

https://www.nvidia.com/en-us/data-center/dgx-2/.

[6] NS3 Network Simulator". [Online]. Available: https://www.nsnam.org.

[7] Mina Tahmasbi Arashloo, Alexey Lavrov, et al. Enabling programmable transport protocols in high-speed NICs. *In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 93-109, 2020.

[8] EunYoung Jeong, Shinae Wood, et al. mtcp: a highly scalable user-level TCP stack for multicore systems. *In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489-502, 2014.

[9] Antoine Kaufmann, Tim Stamler, et al. TAS: TCP acceleration as an OS service. *In Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1-16, 2019.

[10] YoungGyoun Moon, et al. Acceltcp: Accelerating network applications with stateful TCP offloading. *In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77-92, 2020.

[11] RFC 793. [Online]. Available:

https://datatracker.ietf.org/doc/html/rfc793

[12] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. *In Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, IMC ' 10*, page 267–280, New York, NY, USA, 2010. Association for Computing Machinery.

[13] Arjun Roy, Hongyi Zeng, et al. Inside the social network' s (datacenter) network. *In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[14] C.–H. Hsu and U. Kremer, "IPERF: A framework for automatic construction of performance prediction models," *in Workshop on Profile and Feedback–Directed Compilation (PFDC)*, Paris, France. Citeseer, 1998.

[15] G. W. Connery, W. P. Sherer, G. Jaszewski, and J. S. Binder, "Offload of TCP Segmentation to a Smart Adapter," 1999.

[16] Arjun Roy, Hongyi Zeng, et al. Inside the social network' s (datacenter) network. *In Proceedings of the 2015 ACM Conference on Special Interest Group on Data*

Communication, pages 123–137, 2015.

[16] “Checksum Offload.” [Online]. Available: https://www.kernel.org/doc/html/latest/networking/checksum-offloads.html

[17] Kaufmann, Antoine, et al. "TAS: TCP acceleration as an OS service." *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019.

[18] Adam Belay, George Prekas, et al. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. *In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 17.

[19] Cai, Qizhe, et al. "Understanding host network stack overheads." *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 2021.

[20] D. Firestone, A. Putnam, et al., “Azure accelerated networking: Smartnics in the public cloud,” *in 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18),* 2018, pp. 51–66.

[21] YoungGyoun Moon, SeungEon Lee ,et al. Acceltcp:

Accelerating network applications with stateful TCP offloading. *In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20),* pages 77–92, 2020.

[22] M. Ruiz, D. Sidler, et al. "Limago: An fpga-based open-source 100 gbe tcp/ip stack," *in 2019 29th Inter-12 national Conference on Field Programmable Logic and Applications (FPL).* IEEE, 2019, pp. 286–292.

[23] Mina Tahmasbi Arashloo, Alexey Lavrov, et al. Enabling programmable transport protocols in high-speed NICs. *In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 93–109, 2020.

[24] "TCP connection State". [Online]. Available: https://ssup2.github.io/theory_analysis/TCP_Connection_State/

[25] Nginx HTTP Web Server. [Online]. Available: https://nginx:org/en.

[26] Will Glozer. WRK. [Online]. Available: https://github:com/wg/wrk, 2013.

[27] "Ethernet Roadmap 2022". [Online]. Available: https://ethernetalliance.org/technology/ethernet-roadmap/

Abstract

Development of software for TCP offload hardware to reduce host CPU usage for TCP protocols

Seong-min Na
Department of Electrical and
Computer Engineering
The Graduate School
Seoul National University

Network link speeds are being accelerated to meet the modern server demand for high data throughput and low latency. However, supporting the TCP protocol in high-performance networks requires heavy host CPU usage. This paper profiles the host CPU usage of TCP stack to show the seriousness of problem and reviews the solutions of previous papers. Through this, we determine that offloading the TCP stack to hardware would be the most appropriate solution to reduce the host CPU usage in future hundreds of Gbps network.

This paper proposes TOE(TCP Offloading Engine), a TCP offloading hardware for future network. And we

design and develop a TOE Manager that allows applications to use TOE's TCP stack. We tested the end-to-end operation by patching it to the NGINX, the most popular web server. And our evaluation results show that compared to Linux TCP tack, TOE achieves 2.4~3 times higher performance using the same number of cores, or similar performance using 2~4 times less cores.

Keywords : Software development for TCP Offloading
Student Number : 2020-23372