The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel

개요

프로그램 가능한 패킷 처리 기술이 점점 커널 우회 방식을 통해 구현된다. 이 기술에서 사용자 공간 애플리케이션이 네트워크 하드웨어를 직접 제어하여 커널과의 비용이 큰 문맥 전환을 피한다. 그러나 이로 인해 운영 체제의 응용 프로그램 격리, 보안 메커니즘, 구성 및 관리 도구가 작동하지 않는 문제점이 생겼다.

이를 극복하기 위해 eXpress Data Path (XDP)라는 새로운 방식이 제안되었다. Linux 커널 내에서 안전한 실행 환경을 제공하며 사용자 정의 패킷 처리 응용 프로그램을 네트워크 장치 드라이버의 컨텍스트에서 실행할 수 있게 한다. 커널의 일부로 구현이 되어있으며 네트워킹 스택과 함께 작동한다. 응용 프로그램은 C로 구현이 가능하며 커널은 안전성을 위해 정적으로 분석하여 네이티브 명령어로 번역한다.

이 논문에서는 XDP를 통해 L3 routing, DDoS protection, L4 Load balancing를 구현할 수 있으며 싱글 코어에서 초당 최대 2400만 개의 패킷 처리 성능을 보여주고자 한다.

소개

고성능 소프트웨어 패킷 처리의 필요성을 논의한다. 일반 운영 체제의 네트워크 스택이 너무 많은 작업을 수행해 높은 패킷 처리 속도에 대응하기 어렵다는 문제점을 지적한다. 이에 따라 Data Plane Development Kit (DPDK)와 같은 커널 우회 도구가 있지만, 이러한 접근 방식은 기존 시스템과의 통합이 어렵고, 운영 체제의 네트워크 스택 기능을 다시 구현해야 하는 문제가 있다.

이 문제를 해결하기 위해 XDP 방식을 제안한다. XDP는 운영 체제의 네트워크 스택에 직접 프로그래밍을 추가하여 고속 패킷 처리를 지원해 커널 내에서 안전하게 사용자 정의 패킷 처리를 실행할 수 있게 한다. XDP는 기존 시스템과 통합되며, 라우팅 테이블 및 TCP 스택과 같은 커널 기능을 선택적으로 활용할 수 있다. 또한, XDP는 패킷 처리 응용 프로그램을 동적으로 재 프로그래밍할 수 있어 네트워크 트래픽 중단 없이 기능을 추가하거나 제거할 수 있다.

관련 연구

다양한 프레임워크들이 패킷 처리의 프로그래밍 가능성을 제공하였고 여러 응용프로그램들이 생겨났는데, 예를 들어 Switching, routing, name-based forwarding, classification, caching, traffic generation 등 여러 단일 기능 응용 프로그램과 다양한 소스의 패킷을 처리할 수 있는 일반적인 솔루션들이 포함된다.

고속 패킷 처리를 위해 일반적인 운영 체제의 네트워크 스택과의 인터페이스를 제거하는 것이 중요하다고 하며, DPDK와 PF_RING ZC 모듈, Solarflare OpenOnload와 같은 커널 우회 프레임워크가 그 예시인데 이러한 접근 방식은 관리, 유지보수 및 보안 문제가 있다.

XDP는 커널 우회의 반대 접근 방식을 이용해 네트워크 하드웨어의 제어를 커널 내에서 유지하면서 성능 중심의 패킷 처리를 가능하게 한다. 커널과 패킷 처리 코드 간의 경계를 제거하면서 운영 체제의 관리 인터페이스와 보안 보장을 유지하는 장점을 제공한다. XDP의 주요 장점은 커널에서 안전한 실행 환경을 제공하고, 커널 커뮤니티로부터 API 안정선을 보장한다.

XDP Design

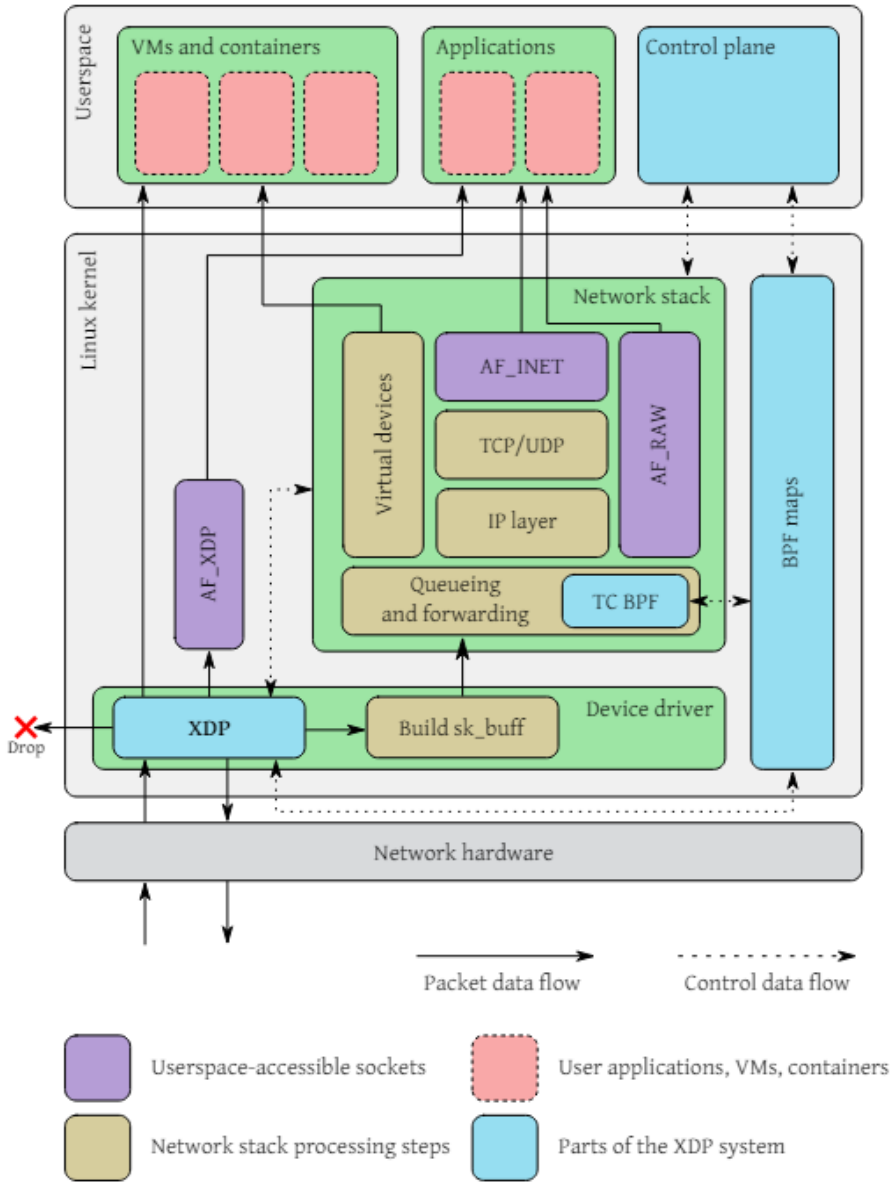

그림 1: XDP가 Linux 네트워크 스택과 통합되는 방식. 패킷 도착 시, 패킷 데이터에 접근하기 전에 장치 드라이버가 메인 XDP 후크에서 eBPF 프로그램을 실행함. 이 프로그램은 패킷을 드롭하거나, 수신한 인터페이스를 통해 다시 전송하거나, 다른 인터페이스로 리다이렉트하거나(가상 머신의 vNIC 포함), 특수 AF_XDP 소켓을 통해 사용자 공간으로 전송하거나, 정규 네트워킹 스택으로 진행하도록 선택할 수 있다. 정규 네트워킹 스택에서는 패킷이 전송을 위해 큐에 들어가기 전에 별도의 TC BPF 후크에서 추가 처리를 수행할 수 있다. 다양한 eBPF 프로그램은 BPF 맵을 통해 서로와 사용자 공간과 통신할 수 있다.

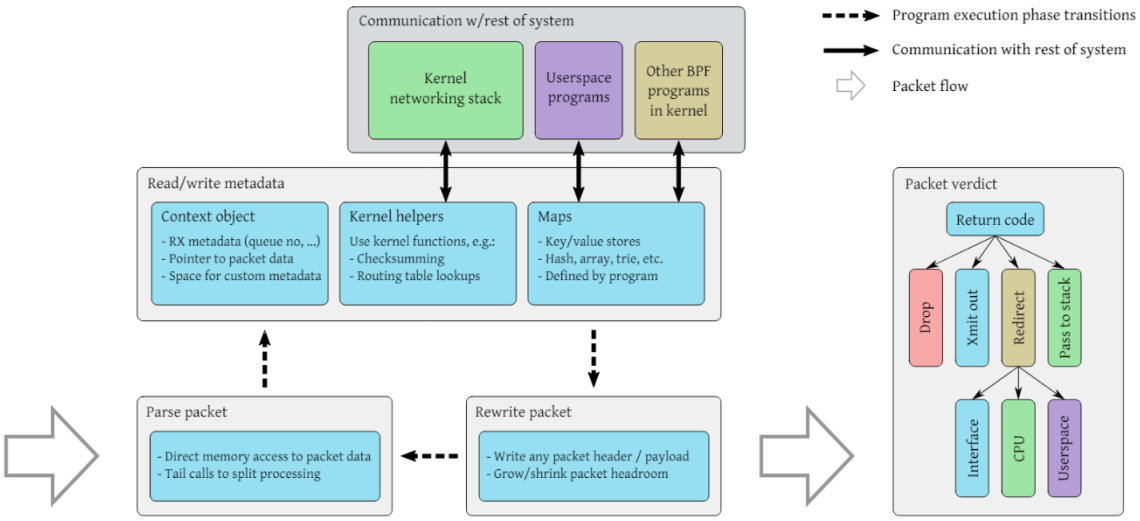

그림 2: XDP 프로그램의 실행 흐름. 패킷이 도착하면 프로그램은 패킷 헤더를 파싱하여 정보를 추출한다. 그런 다음 여러 소스 중 하나에서 메타데이터를 읽거나 업데이트 한다. 마지막으로 패킷을 재작성하고 최종적으로 패킷에 대한 판정이 결정된다. 프로그램은 패킷 파싱, 메타데이터 조회 및 재작성을 번갈아 가며 수행할 수 있으며 모든 과정은 선택적이다. 최종 판정은 프로그램 반환 코드 형태로 결정된다.

여러 버전의 커널을 통해 XDP는 지속적인 피드백, 테스트를 통해 설계가 되었다. 이 논문의 범위 내에서 상세히 다루기 어렵지만 이 섹션에서는 XDP 시스템의 주요 구성 요소들이 어떻게 작동하며, 전체 시스템을 구성하는 방법을 설명한다.

XDP 시스템의 네 가지 주요 구성은 다음과 같다.

1. XDP driver hook: XDP의 main entry point이며, 하드웨어에서 패킷이 도착 했을 때 실행된다.
2. The eBPF virtual machine: XDP program의 byte code를 실행한다. 성능을 향상 시키기 위해 JIT(Just-In-time) 컴파일을 수행한다.
3. BFP maps: 주요 통신 채널로서 키-값 저장소 역할을 하며 시스템의 나머지 부분과의 상호작용을 담당한다.
4. The eBPF verifier: 프로그램이 로드되기 전에 정적으로 프로그램을 검증하여 실행중인 커널을 충돌시키거나 손상시키지 않도록 한다.

XDP drive hook

XDP 프로그램은 네트워크 장치 드라이버의 후크에서 패킷 도착시마다 실행된다. 커널 라이브러리 함수로 구현되어 있어, 패킷 처리는 사용자 공간으로의 문맥 전환 없이 직접 처리된다. 그림 1은 XDP가 Linux 커널에 통합되는 방식을 보여주며, 그림 2는 XDP 프로그램의 실행 흐름을 보여준다.

XDP 프로그램은 패킷 도착 시 context 객체를 통해 실행을 시작하며, 이 객체에는 패킷 데이터에 대한 포인터와 패킷 수신 인터페이스 및 큐에 대한 메타데이터가 포함된다. 프로그램은 보통 패킷 데이터를 파싱으로 시작하고, tail 호출을 통해 다른 XDP 프로그램으로 제어를 전달할 수 있다.

패킷 데이터 파싱 후, XDP 프로그램은 패킷과 관련된 메타데이터를 읽어오고, 추가로 패킷 데이터를 수정하거나 임의의 메타데이터를 첨부할 수 있다. 이러한 작업은 BPF 맵을 통해 시스템의 다른 부분과 통신하고, 다양한 헬퍼 함수를 사용하여 커널 기능을 활용할 수 있다.

처리가 완료되면, XDP 프로그램은 패킷에 대한 최종 결정을 내린다. 이는 패킷을 버릴지, 동일 인터페이스로 다시 전송할지, 커널 네트워킹 스택에서 처리할지, 또는 패킷을 다른 대상으로 리디렉션할지를 결정하는 네 가지 반환 코드 중 하나로 지정된다.

eBPF Virtual Machine

eBPF는 원래의 BPF를 발전시켜 패킷 필터링 응용 프로그램에 광범위하게 사용되는 가상 머신이다. 64비트 레지스터와 11개의 레지스터로 확장되었으며, JIT 컴파일을 통해 네이티브 코드로 변환될 수 있는 기능을 제공한다. 이를 통해 높은 성능의 패킷 처리가 가능해진다. eBPF는 새로운 산술 및 논리 명령어를 포함하여 다양한 명령어가 추가되었으며, 이는 커널 기능과 상호작용할 수 있는 헬퍼 함수 호출을 가능하게 한다. 검증기를 통해 로드되는 프로그램이 커널에 영향을 끼치지 않도록 보장하며, 동적으로 프로그램을 로드하고 재로드 하여 처리량을 관리할 수 있는 유연성을 제공한다.

BPF maps

eBPF 프로그램은 커널 이벤트에 반응하여 실행되며, 각 실행마다 동일한 초기 상태에서 시작하고 프로그램 컨텍스트에서는 지속적인 메모리 저장소에 접근할 수 없다. 대신, 커널은 BPF 맵에 접근할 수 있는 헬퍼 함수를 제공한다. BPF 맵은 eBPF 프로그램이 로드될 때 정의되는 키-값 저장소로, eBPF 코드 내에서 사용될 수 있다. 이 맵은 전역 및

CPU당 변수로 존재하며, 여러 eBPF 프로그램 간에 데이터를 공유하거나 커널과 사용자 공간 간 통신하는 데 사용된다. 맵은 해시 맵, 배열, radix tree 등 다양한 유형이 있으며, 특정 맵 유형은 다른 eBPF 프로그램을 가리키거나 리다이렉트 대상을 포함할 수 있다.

The eBPF Verifier

eBPF 프로그램은 커널 주소 공간에서 직접 실행되며, 잠재적으로 커널 메모리를 접근하거나 손상시킬 수 있다. 이를 방지하기 위해 커널은 모든 eBPF 프로그램을 로딩할 때 정적 분석을 통해 안전하지 않은 동작(예: 임의 메모리 접근)을 방지하고 프로그램이 종료될 것을 보장하는 검증기를 사용한다. 검증기는 프로그램의 제어 흐름을 나타내는 방향성 비순환 그래프(DAG)를 구축하고, 프로그램이 안전한 메모리 접근을 수행하는지 확인한다. 또한, 데이터 접근을 추적하여 프로그램이 안전한지 여부를 결정하며, 안전하지 않은 프로그램은 로딩 과정에서 거부된다.

XDP 프로그램 예제

```c
/* map used to count packets; key is IP protocol, value is pkt count */
struct bpf_map_def SEC("maps") rxcnt = {
.type = BPF_MAP_TYPE_PERCPU_ARRAY,
.key_size = sizeof(u32),
.value_size = sizeof(long),
.max_entries = 256,
};

/* swaps MAC addresses using direct packet data access */
static void swap_src_dst_mac(void *data)
{
unsigned short *p = data;
unsigned short dst[3];
dst[0] = p[0]; dst[1] = p[1]; dst[2] = p[2];
p[0] = p[3]; p[1] = p[4]; p[2] = p[5];
p[3] = dst[0]; p[4] = dst[1]; p[5] = dst[2];
}

static int parse_ipv4(void *data, u64 nh_off, void *data_end)
{
struct iphdr *iph = data + nh_off;
if (iph + 1 > data_end)
return 0;
return iph->protocol;
}
```

```
26
27 SEC("xdp1") /* marks main eBPF program entry point */
28 int xdp_prog1(struct xdp_md *ctx)
29 {
30 void *data_end = (void *)(long)ctx->data_end;
31 void *data = (void *)(long)ctx->data;
32 struct ethhdr *eth = data; int rc = XDP_DROP;
33 long *value; u16 h_proto; u64 nh_off; u32 ipproto;
34
35 nh_off = sizeof(*eth);
36 if (data + nh_off > data_end)
37 return rc;
38
39 h_proto = eth->h_proto;
40
41 /* check VLAN tag; could be repeated to support double-tagged VLAN */
42 if (h_proto == htons(ETH_P_8021Q) || h_proto == htons(ETH_P_8021AD)) {
43 struct vlan_hdr *vhdr;
44
45 vhdr = data + nh_off;
46 nh_off += sizeof(struct vlan_hdr);
47 if (data + nh_off > data_end)
48 return rc;
49 h_proto = vhdr->h_vlan_encapsulated_proto;
50 }
51
52 if (h_proto == htons(ETH_P_IP))
53 ipproto = parse_ipv4(data, nh_off, data_end);
54 else if (h_proto == htons(ETH_P_IPV6))
55 ipproto = parse_ipv6(data, nh_off, data_end);
56 else
57 ipproto = 0;
58
59 /* lookup map element for ip protocol, used for packet counter */
60 value = bpf_map_lookup_elem(&rxcnt, &ipproto);
61 if (value)
62 *value += 1;
63
64 /* swap MAC addrs for UDP packets, transmit out this interface */
65 if (ipproto == IPPROTO_UDP) {
66 swap_src_dst_mac(data);
67 rc = XDP_TX;
68 }
69 return rc;
70 }
```

간단한 XDP 프로그램의 예제이다. 이 프로그램은 패킷 헤더를 파싱하고, 모든 UDP 패킷을 반사하기 위해 송신지 및 수신지 MAC 주소를 교환하여 패킷을 수신한 인터페이스로 다시 보낸다.

1. IP 프로토콜 번호를 키로 사용하여 패킷 처리 횟수를 기록하는 BPF 맵이 정의된다 (라인 1-7). 이 맵은 패킷 수를 업데이트하고, 사용자 공간 프로그램은 이 맵을 폴링하여 XDP 프로그램 실행 중에 통계를 출력할 수 있다.
2. 컨텍스트 객체에서 패킷 데이터의 시작과 끝 포인터를 읽어 직접적인 패킷 데이터 접근을 위해 사용한다 (라인 30-31).
3. 데이터가 데이터 끝 포인터를 초과하지 않도록 확인하여 범위 내에서 데이터를 읽는다 (라인 22, 36, 47). 이러한 점검은 포인터 복사를 통해 올바르게 유지된다 (라인 21-22).
4. 프로그램은 VLAN 헤더를 포함한 패킷 파싱을 스스로 처리해야 한다 (라인 41-50).
5. 패킷 헤더를 수정하기 위해 직접적인 패킷 데이터 접근이 사용된다 (라인 14-16).
6. 커널에서 노출된 맵 조회 도우미 함수를 사용하여 맵 조회를 수행한다 (라인 60). 이는 프로그램 내에서 실제로 호출되는 유일한 함수 호출이다. 다른 모든 함수는 컴파일 시 인라인 처리되며, htons()와 같은 헬퍼 함수도 포함된다.
7. 프로그램 반환 코드를 통해 최종 패킷 결정이 전달된다 (라인 69).

인터페이스에 설치되면 eBPF 바이트 코드로 컴파일되고, 그런 다음 검증기에 의해 확인된다. 검증기가 확인하는 주요 사항은 (a) 루프의 부재 및 프로그램의 총크기, (b) 모든 직접 패킷 데이터 접근이 적절한 경계 검사를 거친후 수행되었는지, (c) 맵 조회 함수에 전달된 매개변수 크기가 맵 정의와 일치하는지, 그리고 (d) 맵 조회의 반환 값이 NULL인지 확인된 후에 접근되는지 등을 확인한다.

성능 평가

이 섹션에서는 XDP의 성능 평가에 대한 결과를 제시한다. 먼저, 고성능 패킷 처리를 위해 DPDK를 기준선으로 삼고, XDP가 제공하는 성능 개선을 Linux 커널 네트워크 스택과 비교한다. 사용된 테스트 환경은 인텔 Xeon E5-1650 v4 CPU를 장착한 시스템으로, Mellanox ConnectX-5 Ex VPI 듀얼 포트 100Gbps 네트워크 어댑터와 TRex 패킷 발생기를 사용한다. 주요 평가 메트릭은 다음과 같다:

1. 패킷 드롭 성능: 들어오는 패킷을 단순히 버리는 작업의 성능을 측정하여 시스템 전체의 오버헤드를 평가한다.
2. CPU 사용량: XDP의 CPU 사용량이 네트워크 부하와 어떻게 변하는지를 측정하여 성능 확장성을 분석한다.
3. 패킷 전달 성능: 패킷 전달이 추가 복잡성을 도입하므로, 이를 평가하여 처리량과 지연 시간을 분석한다.

결과적으로, XDP는 최소 크기의 패킷을 사용하여 초당 처리 가능한 패킷 수를 측정하고, CPU 코어 수를 증가시키면서 성능이 어떻게 변하는지를 분석하였다. 이러한 평가를 통해 XDP가 다른 시스템에 비해 어떤 성능을 보이는지를 자세히 살펴보았다.

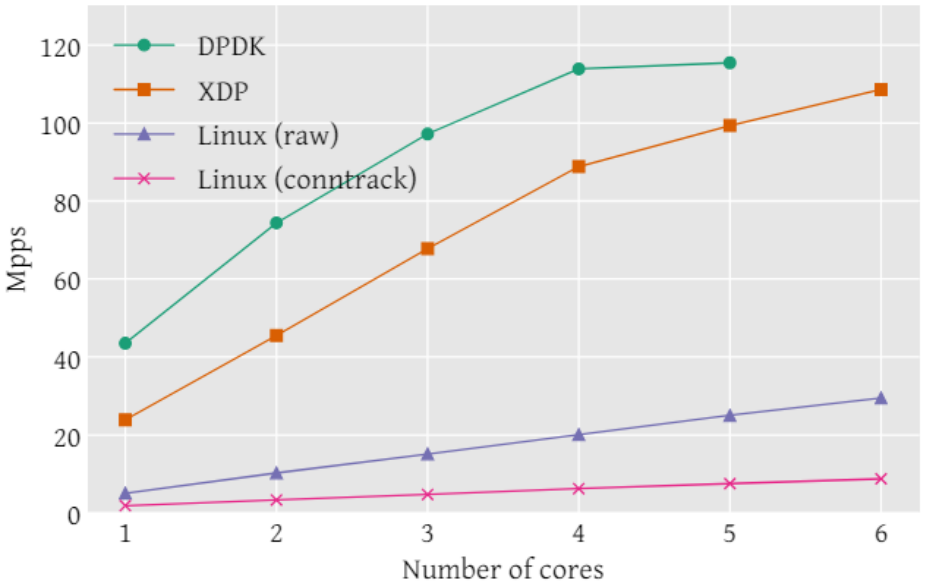

그림 3: 패킷 드롭 성능. DPDK는 제어 작업에 하나의 코어를 사용하므로 패킷 처리에는 5개의 코어만 사용 가능하다.

그림 3는 다양한 시스템의 패킷 드롭 성능을 보여주는 그래프이다. XDP는 단일 코어에서 24 Mpps의 기본 성능을 보이며, DPDK는 43.5 Mpps를 기록한다. 이들은 모두 성능이 코어 수에 비례하여 선형적으로 증가하다가 PCI 버스의 성능 한계에 도달하면 115 Mpps에서 멈추는 것으로 나타났다.

또한 그림은 Linux 네트워킹 스택의 성능도 보여준다. iptables의 "raw" 테이블을 사용한 모드는 4.8 Mpps의 성능을 보이며, conntrack 모듈을 사용한 모드는 1.8 Mpps에서 성능이 시작되어 오버헤드가 크게 증가한다.

XDP의 경우, 아무 동작도 하지 않고 패킷 카운터를 업데이트하고 패킷을 스택으로 전달하는 프로그램을 설치했을 때, 성능이 4.5 Mpps로 떨어지며 처리 오버헤드는 13.3 ns에 이른다는 것도 확인했다.

이와 같은 결과는 XDP가 네트워크 처리에서 뛰어난 성능을 발휘하며, 특히 많은 수의 패킷을 빠르게 처리해야 하는 경우에 유용하다는 것을 보여준다.

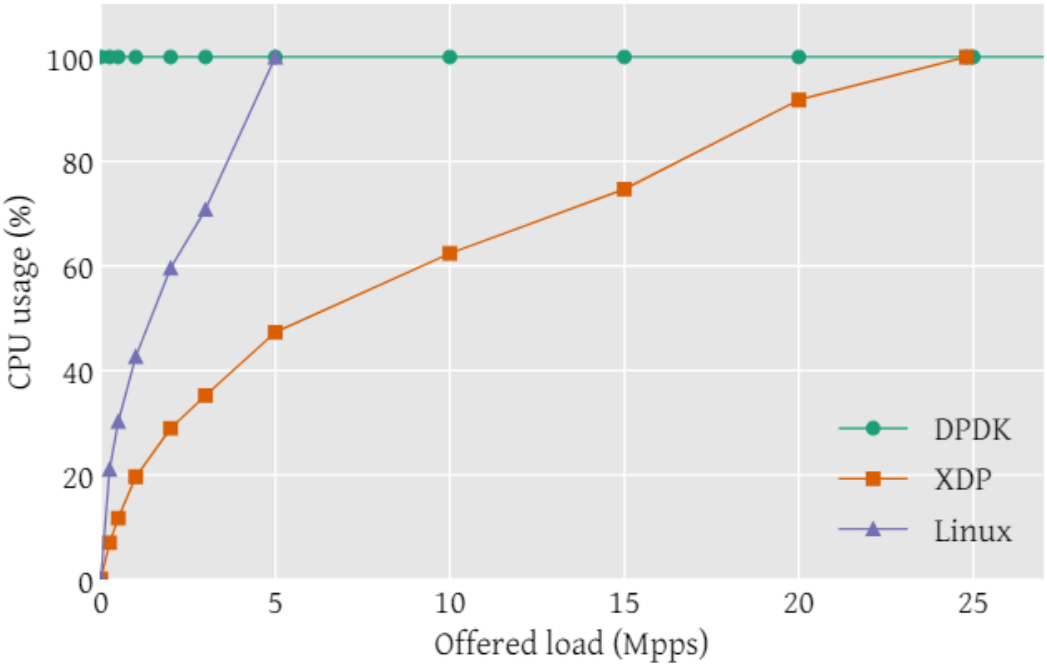

그림 4: 드롭 시나리오에서의 CPU 사용량. 각 선은 해당 방법의 최대 처리 능력에서 멈춘다. DPDK 선은 Figure 3에 표시된 최대 성능까지 100%로 유지된다.

각 시스템이 단일 CPU 코어에서 패킷 드롭 응용 프로그램을 실행할 때 CPU 사용량을 mpstat 시스템 유틸리티를 사용하여 측정했다. 결과는 그림 4에 나타나 있다. 테스트는 각 시스템이 단일 코어에서 처리할 수 있는 최대 패킷 부하까지 제공한다.

DPDK는 패킷 처리에 전용 코어를 할당하고 바쁜 폴링(busy polling)을 사용하기 때문에 CPU 사용률이 항상 100%에 가깝다. 이는 그림 상단의 녹색 선으로 표시된다. 반면, XDP와 Linux는 제공된 부하에 따라 CPU 사용량이 부드럽게 조절되며, 낮은 부하 수준에서 상대적으로 더 큰 CPU 사용량 증가가 있다.

그래프의 왼쪽 하단에서 비선형성은 인터럽트 처리의 고정된 오버헤드 때문이다. 낮은 패킷 속도에서는 각 인터럽트 당 처리되는 패킷 수가 적어져서 각 패킷 당 더 높은 CPU 사용량이 발생한다.

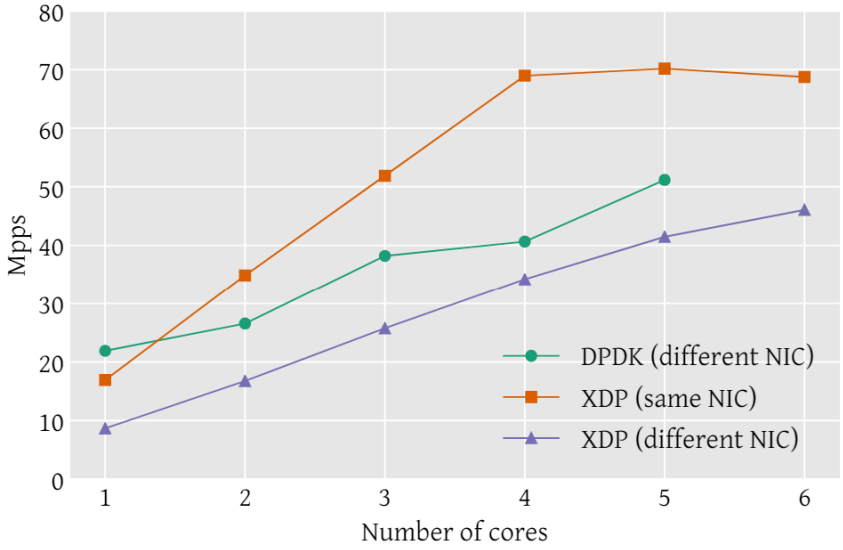

그림 5: 패킷 전달 처리량. 동일 인터페이스에서의 송수신은 동일한 PCI 포트에서 더 많은 대역폭을 사용하므로, 우리는 70 Mpps에서 PCI 버스 한계에 도달한다.

그림 5는 패킷 전달 성능을 보여준다. 전달 응용 프로그램들은 패킷을 전달하기 전에 수신된 패킷의 출발지와 목적지 주소를 교환하는 간단한 이더넷 주소 변경을 수행한다. 이는 패킷 전달이 작동하기 위해 필요한 최소한의 변경 작업이므로, 이 결과는 전달 성능의 상한선을 나타낸다.

	Average		Maximum		$< 10\mu s$	
	100 pps	1 Mpps	100 pps	1 Mpps	100 pps	1 Mpps
XDP	$82\mu s$	$7\mu s$	$272\mu s$	$202\mu s$	0%	98.1%
DPDK	$2\mu s$	$3\mu s$	$161\mu s$	$189\mu s$	99.5%	99.0%

표 1: 패킷 전달 지연 시간. 측정 기계는 동일한 NIC의 두 포트에 연결되어 있으며, 높고 낮은 패킷 속도 (100 pps 및 Mpps)에서 50초 동안의 종단 간 지연 시간을 측정함.

그림 5는 패킷 전달 성능을 나타내며, 여기에는 패킷의 출발지와 목적지 MAC 주소를 교환하는 간단한 이더넷 주소 재작성이 포함된다. 이는 패킷 전달에 필요한 최소한의 재작성이며, 결과는 전달 성능의 상한을 나타낸다. XDP는 동일 NIC와 다른 NIC를 통해 패킷을 전달할 수 있는 두 가지 운영 모드를 포함하여 그래프에 표시된다. 반면 DPDK 예제 프로그램은 다른 인터페이스를 통해 패킷을 전달하는 방식만 지원하므로 해당 운영 모드만 테스트에 포함되었다. Linux 네트워킹 스택은 이 최소 전달 모드를 지원하지 않으며, 전체 브리징 또는 라우팅 조회가 필요하므로 결과가 직접적으로 비교되지 않는다. 그림 5는 CPU 코어 수에 따라 선형적으로 확장되는 성능을 보여주며, XDP는 패킷을 동일 인터페이스에서 수신하고 전달할 때 성능이 크게 향상되는 것을 나타낸다.

토론

XDP는 일반적인 Linux 네트워킹 스택보다 높은 성능을 보인다. 그러나 대부분의 사용 사례에서는 DPDK의 성능에는 약간 못 미친다. 이는 DPDK가 코드의 가장 낮은 수준에서 더 많은 성능 최적화를 포함하기 때문이다. 예를 들어 패킷 드롭 시나리오에서 XDP는 단일 코어에서 24Mpps의 처리를 보이며, 이는 패킷당 약 41.6 ns에 해당한다. 반면에 DPDK는 43.5 Mpps로 패킷당 약 22.9 ns를 달성한다. 이러한 18.7 ns의 차이는 테스트 시스템의 3.6 GHz 프로세스에서 약 67 클럭 사이클을 차지한다. 이는 마이크로 최적화의 중요성을 강조한다. 예를 들어 테스트 시스템에서 하나의 함수 호출이 약 1.3 ns의 오버헤드를 가지고 있음을 측정했다. mlx5 드라이버는 패킷 처리 시 10개의 함수 호출을 수행하며, 이는 XDP와 DPDK 간의 성능 차이 중 약 13 ns를 차지한다.

이러한 오버헤드 중 일부는 Linux와 같은 범용 운영 체제에서 피할 수 없다. 디바이스 드라이버와 서브시스템은 다양한 시스템과 구성을 지원하기 위해 구조화되어 있다. 그러나 일부 최적화가 가능한데 예를 들어 특정 하드웨어에서 필요하지 않은 DMA 관련 함수 호출을 제거한 실험을 통해 패킷 드롭 성능을 29 Mpps로 향상시켰다. 최적화를 통해 XDP와 DPDK 간의 성능 격차가 시간이 지남에 따라 축소될 수 있다. 다른 드라이버들에서도 관찰되는데 예를 들어 Intel 40 Gbps 카드용 i40e 드라이버는 XDP와 DPDK 모두 NIC 하드웨어 성능 한계까지 완벽한 성능을 달성한다.

이러한 점들을 고려해 XDP가 DPDK와의 성능 차이를 줄일 가능성이 있다고 볼 수 있다. 그러나 XDP는 시스템의 유연성과 통합 장점을 고려할 때 이미 많은 사용 사례에서 유용한 선택지임을 보여준다.

실제 사용 사례

XDP의 다양한 측면을 활용하여 유용한 응용 프로그램을 구현하는 방법을 보여주기 위해, 이 섹션에서는 세 가지 예제 사용 사례를 설명한다. 이 사용 사례들은 어떤 형태로든 배포되어 사용되었으며, 우리는 코드를 제공할 수 있도록 간소화된 버전을 사용한다. 또한 독립적으로 실제 네트워크 서비스를 구현하는 과정에서 마주친 몇 가지 문제를 다루는 [38]을 참고하도록 한다.

이 섹션의 목적은 각 사용 사례를 XDP로 구현 가능성을 보여주는 것이므로, 최첨단 구현과의 철저한 성능 평가는 수행하지 않는다. 대신 일반적인 Linux 커널 스택을 간단한 성능 기준으로 삼고, XDP 응용 프로그램을 그에 대비하여 벤치마크한다.

이 세 가지 사용 사례는 소프트웨어 라우터, 인라인 DoS(Denial of Service) 방어 응용 프로그램, 그리고 레이어-4 로드 밸런서이다.

1. Software Routing

   XDP는 리눅스 커널의 기능을 이용해 라우팅 테이블 조회를 효율적으로 수행하며, 이는 완전한 BGP 테이블을 포함한 라우팅 작업에서 높은 성능을 제공한다. XDP는 라우팅 결정을 위해 패킷을 즉시 전달할 수 있고, 필요시 네트워킹 스택으로 전환하여 이웃을 해결한다. 이를 통해 XDP는 단일 코어에서도 10 Gbps 링크에서 라우터를 효율적으로 운영할 수 있는 성능을 보여준다.

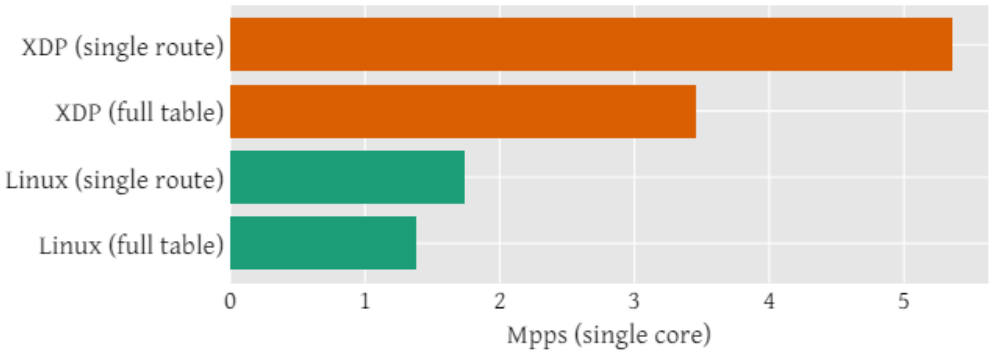

   그림 6: 소프트웨어 라우팅 성능. 성능이 코어 수에 선형적으로 스케일되므로, 단일 코어 결과만 표시된다.

2. Inline Dos Mitigation

   XDP를 사용한 DoS 공격 완화 방식은 클라우드 플레어의 Gatebot 아키텍처를 기반으로 한다. 이 방식은 분산 포인트 오브 프레젠스(PoP)에 위치한 서버에서 트래픽을 샘플링하여 중앙에서 분석하고, 분석 결과를 기반으로 완화 규칙을 생성하여 엣지 서버에 배포한다. XDP 프로그램은 이러한 규칙을 기반으로 패킷을 필터링

하고, 공격 패킷을 드롭하며, 일치 카운터를 업데이트한다. 실험 결과에 따르면, XDP 필터를 사용하면 단일 CPU 코어에서도 10 Gbps의 최소 패킷 DoS 트래픽을 효과적으로 처리할 수 있으며, 이는 특별한 하드웨어나 애플리케이션 변경 없이도 유연하게 구현할 수 있는 장점을 제공한다.

그림 7: DDoS 성능. 서버에 직접 공격 트래픽이 증가함에 따라 초당 TCP 트랜잭션 수.

3. Load Balancing

CPU Cores	1	2	3	4	5	6
XDP (Katran)	5.2	10.1	14.6	19.5	23.4	29.3
Linux (IPVS)	1.2	2.4	3.7	4.8	6.0	7.3

표 2: 로드 밸런서 성능(Mpps).

로드 밸런서 사용 사례에서는 Facebook의 Katran 로드 밸런서의 XDP 구성 요소를 사용하여 서비스용 IP 주소를 발표하고, 패킷을 해싱하여 대상 응용 프로그램 서버를 선택한 후 캡슐화하여 전송한다. XDP 프로그램은 이러한 작업을 수행하고, BPF 맵에 구성 데이터를 저장하여 전적으로 eBPF 프로그램에서 캡슐화를 구현한다. 테스트 결과, XDP는 IPVS에 비해 CPU 개수에 따라 선형적으로 스케일링되며 성능이 4.3배 향상되었다.

XDP 미래 방향

위에서 보았듯이, XDP는 높은 성능을 제공하며 다양한 실제 사용 사례를 구현하는 데 사용될 수 있다. 그러나 이는 XDP가 완성된 시스템이라는 의미는 아니다. 오히려 Linux 커널의 일부로서 XDP는 지속적인 개선을 거듭하고 있다. 이 개발 노력 중 일부는 XDP가 일반 목적 운영 체제 커널에 점진적으로 통합되면서 발생하는 문제를 완화하는 데 집중되고 있다. 다른 노력들은 XDP 능력의 경계를 더욱 넓히기 위해 계속되고 있다. 이 섹션에서는 이러한 노력 중 일부를 논의한다.

1. eBPF 프로그램의 한계

   eBPF 가상 머신에 로드되는 프로그램은 eBPF 검증기에 의해 분석되며, 두 가지 주요 제한 사항이 존재한다. 첫째, 프로그램이 종료됨을 보장하기 위해 루프를 금지하고 프로그램의 최대 크기를 제한한다. 둘째, 메모리 접근의 안전을 보장하기 위해 레지스터 상태 추적을 통해 이루어진다. 검증기는 커널의 안전성을 최우선으로 하여, 안전성을 증명할 수 없는 프로그램은 거부할 수 있다. 최근에는 함수 호출 지원이 추가되었고, 경계가 있는 루프의 지원이 계획 중에 있으며, 검증기의 효율성 개선이 진행 중이다. 또한, eBPF 프로그램은 커널 라이브러리의 표준 라이브러리 기능이 없어 일부 제한이 있지만, 커널의 라이프 사이클 관리 및 헬퍼 함수를 통해 일부 완화될 수 있다. 하나의 네트워킹 인터페이스에는 하나의 XDP 프로그램만 연결할 수 있지만, tail call 기능을 사용하여 프로그램 간 협력이 가능하다.
2. User Experience와 Debugging

   XDP 프로그램은 커널에서 실행되기 때문에 일반 사용자 공간 프로그램의 디버깅 도구는 적용할 수 없다. 대신, 커널의 tracepoints, kprobes, perf 서브시스템의 성능 카운터 등을 사용하여 디버깅 및 내부 검사를 수행할 수 있다. 그러나 이러한 커널 특화 도구는 커널 생태계에 익숙하지 않은 개발자에게는 제한적일 수 있다. 이를 보완하기 위해 BPF 컴파일러 컬렉션, bpftool 내부 검사 프로그램, libbpf 라이브러리 등의 도구가 존재하며, 이들 도구는 계속해서 개선되고 있다.
3. Driver Support

   XDP 프로그램을 실행하기 위해 각 장치 드라이버는 핵심 네트워킹 스택에서 노출된 API를 지원해야 한다. 점점 더 많은 드라이버가 지원을 추가하고 있지만, 사용 사례에 필요한 기능을 완전히 지원하는 하드웨어 선택이 중요하다.
4. 성능 개선

   XDP와 DPDK 사이의 성능 차이를 줄이기 위한 지속적인 노력이 진행 중이며, 예로 driver code 마이크로 최적화, core XDP 코드에서 필요 없는 연산 제거, batching을 통한 processing cost amortization 등이 있다.

5. QoS와 Rate transitions

   현재 XDP는 다양한 QoS 수준을 지원하지 않으며, 특히 네트워크 특성의 불일치에 따른 백프레셔를 처리하지 않는다. XDP에 QoS 기능은 없지만 Linux 커널에서는 Active Queue Management (AQM)과 패킷 스케줄링 알고리즘을 제공한다. QoS 및 AQM에 대한 지원을 개선하기 위해 네트워킹 스택의 기능 일부를 선택적으로 통합하는 가능성을 탐구하고 계획 중이다.

6. Transport Protocol 가속

   XDP를 사용하여 높은 속도의 패킷 처리를 협력적으로 운영 체제에 통합함으로써 기존 운영 체제 기능을 활용하면서 처리를 가속화할 수 있음을 보였다. XDP는 상태를 가지지 않는 패킷 처리에 초점을 맞추고 있지만, TCP와 같은 상태를 가지는 전송 프로토콜에 동일한 모델을 확장함으로써 신뢰성 있는 전송을 필요로 하는 응용 프로그램에도 많은 성능 이점을 제공할 수 있을 것이다.

7. Zero-copy to userspace

   XDP 프로그램은 사용자 공간 응용 프로그램이 열어 놓은 특수 소켓으로 데이터 패킷을 리디렉션할 수 있음을 언급한 바 있다. 이는 로컬 머신에서 실행되는 네트워크 중심 응용 프로그램의 성능을 개선하는 데 사용될 수 있다. 그러나 초기 구현에서는 여전히 패킷 데이터를 복사하는 방식으로 작동하여 성능에 부정적인 영향을 미칠 수 있다.

8. XDP building block

   DPDK가 상위 수준 패킷 처리 프레임워크의 저수준 구성 요소로 사용되는 것처럼, XDP는 상위 수준 응용 프로그램을 위한 런타임 환경으로 기능할 수 있다. Cilium 보안 미들웨어, Suricata 네트워크 모니터, Open vSwitch 및 P4-to-XDP 컴파일러 프로젝트와 같은 프레임워크와 응용 프로그램이 XDP를 활용하고 있으며 XDP를 DPDK의 저수준 드라이버로 추가하는 노력도 진행 중이다.

결론

XDP를 소개하고, 이 시스템이 운영 체제 커널에 빠르고 프로그래밍 가능한 패킷 처리를 안전하게 통합하는 방법을 설명한다. XDP는 단일 CPU 코어에서 최대 24 Mpps의 원시 패킷 처리 성능을 보여주며, 커널 보안과 관리 호환성 유지, 필요에 따른 기존 커널 스택 기능의 선택적 활용, 안정적인 프로그래밍 인터페이스, 애플리케이션에 대한 완전한 투명성 등의 매력적인 기능을 제공한다. XDP는 서비스 중단 없이 동적으로 재프로그래밍할 수 있으며, 특수 하드웨어나 패킷 처리 전용 리소스를 필요로 하지 않는다. 이러한 특성으로 인해 XDP는 커널 바이패스 솔수션에 비해 매우 유망한 대안으로 자리잡고 있다.