ETRI

2015
Electronics and
Telecommunications
Trends

10Gbps 네트워크 트래픽 저장 및 실시간 인덱스 생성

10Gbps Network Traffic Capture and Runtime Indexing

최선오 (S. Choi) 네트워크보안연구실 선임연구원
이주영 (J.Y. Lee) 네트워크보안연구실 선임연구원
최양서 (Y.S. Choi) 네트워크보안연구실 책임연구원
김종현 (J.H. Kim) 네트워크보안연구실 책임연구원
김익균 (I.K. Kim) 네트워크보안연구실 실장

요즘 3.20 대란이나 한수원 침해사고 같은 각종 사이버 공격이 빈번하게 발생하고 있다. 이러한 사이버 공격에 대응하기 위하여 네트워크 트래픽을 수집 · 저장하고 사이버 공격분석에 사용하려는 다양한 노력이 행해지고 있다. 그러나 일반적으로 10Gbps 같은 고속 네트워크에서 네트워크 트래픽을 수집 · 저장하는 것은 쉬운 일이 아니다. 그래서 이 문건에서는 10Gbps 네트워크 트래픽 수집 · 저장에 관한 기술동향을 다루고 이어서 수집된 대용량 트래픽을 효율적으로 검색하기 위하여 비트맵 인덱스를 생성하는 다양한 방법을 소개하고 마지막으로 비트맵 인덱스를 사용한 효율적인 검색방법에 대해 소개한다.

Ⅰ. 서론

인터넷이 발달함에 따라 인류는 많은 유익을 누리고 있다. 웹에서 우리는 다양한 정보를 제공받을 수 있고 스마트폰이 발달함에 따라 언제 어디서나 메일을 확인하고 보내고 물건을 구매하고 뉴스를 시청할 수 있는 편리함을 누리고 있다.

그러나 인터넷의 발달 이면에 우리는 또 다른 문제에 직면해있다. 경제적인 이득을 취하거나 사회의 혼란을 일으키기 위하여 해커들의 다양한 공격이 기존의 인터넷 시스템에 행해지고 있다. 예를 들어 2013년에 일어난 3.20 대란은 수많은 컴퓨터를 작동불능의 상태로 만들어버렸고 2014년 말에 발생한 한수원 침해사고는 원전시스템파괴라는 위협을 주고 사회에 불안감을 높이고 있다.

이러한 문제에 대응하기 위하여 Intrusion Detection System(IDS)/ Intrusion Prevention System(IPS)가 사용되고 있지만, IDS/IPS의 경우에는 기존의 알려진 공격에 대해서만 대응할 수 있는 단점이 존재한다. 이러한 IDS/IPS의 문제점을 보완하기 위하여 네트워크 플로우 데이터를 저장하여 분석하려는 노력들이 있지만, 네트워크 플로우만으로는 분석에 제한이 있다. 이러한 어려움을 극복하기 위하여 네트워크 트래픽을 저장하고 저장된 트래픽을 분석에 사용하려는 노력들이 행해지고 있다.

우리는 TCPDUMP나 WireShark를 사용하여 네트워크 트래픽을 수집할 수 있다. 그러나 이것들은 제한된 대역폭을 가지는 네트워크에서만 행해질 수 있고 기가비트 대역폭을 가지는 최신의 네트워크에서 네트워크 트래픽을 수집하는 것은 쉬운 일이 아니다. 우리는 다음 단락에서 10Gbps 네트워크상에서 네트워크 트래픽을 저장하는 기술의 최신동향에 대해 알아볼 것이다.

네트워크상에서 수집되는 주요데이터는 크게 두 가지로 구분된다. 첫 번째는 네트워크 플로우 데이터이고 두 번째는 네트워크 패킷 데이터이다. 네트워크 플로우 데이터의 경우에는 nfdump 같은 도구를 사용하여 검색할 수 있고 패킷 데이터의 경우에는 tcpdump 같은 도구를 사용하여 검색할 수 있다. 그러나 여기에서 말하는 검색은 linear scan이기 때문에 많은 데이터가 있을 시에는 매우 비효율적이다.

예를 들어 10Gbps 네트워크의 경우에는 초당 대략적으로 1GB의 패킷 데이터가 저장된다. 그리고 최악의 상황을 가정할 경우 하루에 약 86TB의 데이터가 저장된다. 이러한 대용량 데이터에 대하여 linear scan을 하는 것은 현실적으로 불가능하다. 따라서 대용량 네트워크 데이터에 대하여 효율적으로 검색하기 위하여 인덱스가 필요하다.

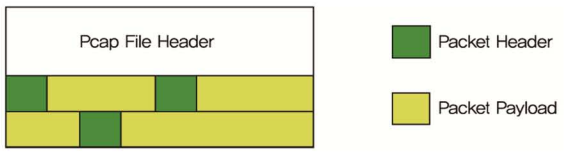
우리는 인덱스를 위하여 데이터베이스를 사용할 수 있다. 일반적으로 데이터베이스의 경우에는 초당 5만 건 정도의 데이터를 기록할 수 있는 것으로 알려져 있다. 그러나 10Gbps 네트워크의 경우에는 초당 수백만 개의 패킷이 들어오게 된다. 따라서 데이터베이스를 이용하여 10Gbps 네트워크 데이터에 대하여 인덱스를 생성하는 것은 현실적으로 불가능하다.

일반적으로 인덱스를 위하여 B-tree를 이용할 수 있다. 그러나 B-tree의 경우에는 데이터의 생성, 변경, 삭제가 이루어지는 경우에 사용되고 우리가 목표로 하는 네트워크 데이터의 경우에는 데이터의 변경이나 삭제 없이 데이터의 생성만 이루어지는 특수한 데이터가 된다. 이러한 데이터의 경우에는 Bitmap Index를 활용하는 것이 효율적이라고 알려져 있다.

이 문건의 구조는 다음과 같다. Ⅱ장에서는 네트워크 패킷 저장구조에 대해서 다루고 Ⅲ장에서는 10Gbps 네트워크 트래픽 저장을 위한 동향을 다룬다. 그리고 Ⅳ장에서는 비트맵 인덱스 생성 및 검색 기법에 대한 동향을 다루고 마지막으로 Ⅴ장에서 결론을 준다.

Ⅱ. 네트워크 패킷 저장

패킷 저장을 위한 표준은 Berkeley Packet Fil-



(그림 1) Pcap File Format[3]

ter(BPF)[1]를 사용하는 것이다. BPF는 libpcap[2] 라이브러리에서 사용된다. BPF 필터는 라이브 트래픽을 필터링하거나 저장된 트래픽을 필터링하는 곳에도 사용될 수 있다. 그러한 트래픽들은 pcap 포맷으로 저장된다. BPF가 패킷 필터링을 위한 표준이기 때문에 pcap이 파일기반 패킷 저장을 위한 표준이 된다.

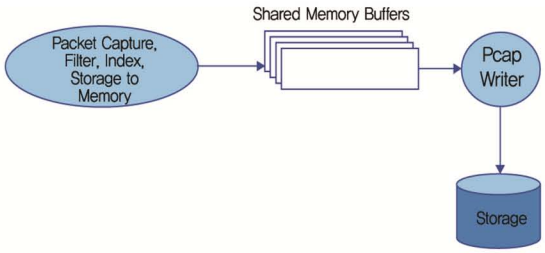
pcap 파일은 (그림 1)과 같이 패킷이 수집되는 인터페이스 종류와 패킷길이를 포함하는 헤더를 가지고 있다. 헤더 뒤에 각각의 패킷이 저장된다. 각 패킷은 타임스탬프와 두 개의 길이를 포함하는 헤더를 가지고 있다. 첫번째 것은 패킷의 길이이고 두 번째 것은 저장되는 데이터의 길이이다. 헤더는 인덱스 정보를 포함하고 있지 않기 때문에 특정 패킷을 읽기 위하여 그 패킷에 도달할 때까지 pcap 파일을 처음부터 읽어야 한다.

Ⅲ. 10G 트래픽 저장

10Gbps 네트워크 트래픽을 일반장비로 저장하는 것에 관한 이슈를 다룬 논문은 n2disk[3]이다. n2disk는 2가지 방식으로 동작하는데 첫 번째는 single thread를 사용하는 것이고 두 번째는 multi thread를 사용하는 것이다. 첫 번째 것은 기가비트 네트워크 트래픽을 수집하는데 적합하고 두 번째 것은 10Gbps 네트워크를 위해 사용된다.

1. Single Thread 시스템

Single Thread 시스템에서는 (그림 2)와 같이 단 하나의 packet consumer thread가 있다. 패킷리더는 하나의 네트워크 인터페이스로부터 패킷을 읽고 필터링 조건을 만족하지 않는 패킷은 드롭시킨다. 그리고 필터



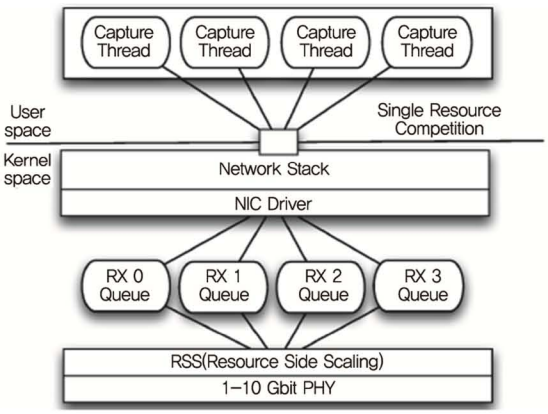
(그림 2) Single Thread n2disk System[3]

링 조건이 만족되는 패킷들은 메모리 버퍼로 복사한다.

이 버퍼들은 pcap file header들로 미리 채워진다. 그리고 수집된 패킷들은 해당 pcap 패킷헤더를 가진 버퍼로 복사된다. 메모리 버퍼가 가득차면 리더는 다음 버퍼를 채우기 시작한다. 메모리 버퍼를 디스크에 쓰는 것은 두 번째 스레드에 의해 수행된다. write() 같은 운영체제에 의한 데이터 버퍼링 오버헤드를 줄이기 위하여 n2disk는 direct I/O를 이용하였다. 이것은 malloc() 대신에 posix_memalign()을 사용하여 공유 메모리 버퍼를 할당함으로써 수행된다. lock-less design을 위하여 writer와 reader는 분리된다. 기록할 새로운 버퍼가 준비되었는지 writer에게 알려주기 위하여 wait()/signal()를 사용하는 대신에 writer는 1 usec sleep을 하고 폴링을 한다. 패킷들은 리눅스에서는 PF_RING[4]을 사용하여 수집될 수 있다.

가. PF_RING의 필요성

최신 멀티코어지원 네트워크 어댑터는 몇 개의 RX/TX 큐들로 논리적으로 분리된다. 그 큐들에서 패킷들은 Intel I/O Acceleration Technology(I/O AT)[5][6]의 Receive-Side Scaling(RSS) 같은 하드웨어 기반 장비를 사용하여 부하를 분산한다. 하나의 큐를 몇개의 더 작은 큐로 나눔으로써 코어들에게 부하를 분산하여 성능을 향상시킬 수 있다. 최신 네트워크 인터페이스 카드들은 정적으로 또는 동적으로 변경 가능한 분산 정책을 지원한다[7]. 이용 가능한 큐의 수는 Network Interface Card(NIC) 칩셋에 따라 다르고 이용 가능한 시스템 코



(그림 3) Design Limitation in Network Monitoring Architecture[4]

어의 수에 의해 제한된다.

그러나 대부분 운영체제에서 패킷들은 단 하나의 RX 큐를 가지는 멀티코어 이전 시대에 설계된 패킷폴링 기술을 사용하여 전달된다. 운영체제의 관점에서 100Mbit 카드와 10Gbit 카드는 차이가 없다. (그림 3)과 같이 디바이스 드라이버는 단 하나의 큐를 가지는 것처럼 모든 큐를 하나로 합친다. 그리고 이러한 디자인은 아무리 애플리케이션이 몇 개의 스레드를 사용하여 패킷을 처리한다고 하더라도 성능제한의 주요한 이유가 된다.

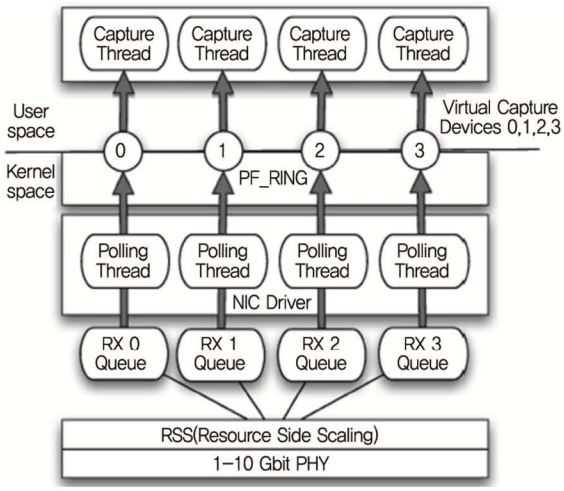
메모리 복사의 수를 줄이기 위하여 패킷을 커널영역에서 사용자 영역으로 옮길 때 표준 시스템 콜을 대신에 메모리 매핑에 기반한 zero-copy 기술을 사용한다[8]. 커널 내에서 패킷은 NIC 드라이버 계층에서 시작한다. 그곳에서 들어오는 패킷들은 네트워크 스택에 의해 처리될 때까지 socket buffer라고 불리는 임시 메모리에 복사된다[9][10]. 네트워크 모니터링에서 패킷들은 라우팅이나 관리를 위해 사용되지 않는 특정 어댑터에서 수신되기 때문에 소켓 버퍼의 할당이나 재할당은 불필요하고 zero-copy가 네트워크 계층이 아니라 드라이버 계층에서 직접적으로 시작할 수 있다.

한편 메모리 대역폭은 캐시구조가 잘못 이용될 때 낭비될 수 있다. 인터럽트 요청(IRQ)을 적절하지 못하게 분산시키는 것은 과도한 캐시실패를 일으킬 수 있다. 이 문제를 해결하기 위하여 인터럽트 요청 핸들러와 캡처 스레드는 같은 프로세서나 코어에서 수행되어야 한다. 그러나 불행하게도 대부분 운영체제는 인터럽트 요청을 코어들 간에 단순하게 분산시킨다. 이것은 실제로 패킷 손실의 일반적인 경우가 된다. 최신의 운영체제들은 인터럽트 요청 분산 정책을 사용자가 수정할 수 있게 한다[11]. 그러나 아쉽게도 현재의 운영체제들은 큐 아이디를 사용자 공간으로 전달하지 못하기 때문에 애플리케이션들은 CPU를 적절하게 설정한 정보를 갖지 못한다.

요약하면 병렬처리를 제한하는 두 가지 이슈가 있다. 첫째는 같은 소켓으로부터 오는 패킷들을 동시에 소비하려고 하는 멀티스레드 애플리케이션들의 경쟁문제이고 둘째는 불필요한 패킷복사, 부적절한 스케줄링과 인터럽트 밸런싱이 메모리 대역폭 사용문제를 일으키는 것이다.

나. PF_RING의 설계

PF_RING은 멀티큐 어댑터와 멀티코어 프로세서를 이용하는 고성능 패킷수집 기술이다. 그것은 멀티 스레드 폴링과 zero-copy 기술을 사용한다. PF_RING은 멀티큐를 지원하고 (그림 4)와 같이 그것들을 사용자들에게 가상의 수집장치로 인식되게 한다. 가상의 수집장치



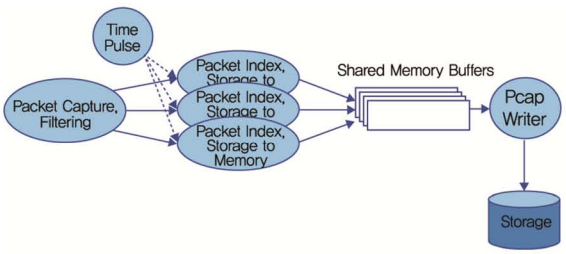
(그림 4) Multi-queue Aware Packet Capture Design[4]

는 애플리케이션이 몇 개의 독립적인 스레드로 나누어지게 하고 각각의 스레드는 트래픽의 일부분을 받고 처리한다. 애플리케이션은 모든 큐로부터 패킷을 받기 위하여 물리장치(예, eth1)에 바인드될 수도 있고 또는 특별한 큐로부터 패킷을 받기 위하여 가상장치(예, eth1@2)에 바인드될 수도 있다. PF_RING은 read()와 같은 비싼 시스템 콜을 사용하지 않고 커널영역에서 사용자영역으로 패킷을 옮기기 위하여 메모리 매핑에 기반한 zero-copy 기술을 사용한다.

2. Multi Thread 시스템

10Gbps 네트워크에서 n2disk의 싱글 스레드 버전은 제한된 패킷 필터링 규칙을 사용하는 고성능 CPU (3.0GHz 이상)에서만 성능을 유지할 수 있다. 이유는 패킷을 처리하는 데 요구되는 CPU 사이클의 수가 싱글 코어에 의해 주어지는 사이클 수보다 크기 때문이다. 예를 들어 타임스탬핑은 소프트웨어적으로 처리하기에 비싼 연산인데 약 80 tick을 요구한다. 그리고 2GHz 코어에서 134 tick가 이용 가능한 것을 생각할 때 타임스탬핑은 비싼 연산이 된다. 그러므로 하드웨어적으로 타임스탬핑을 지원하지 않는 시스템의 경우 (그림 5)와 같이 멀티스레드 시스템을 사용해야 한다.

n2disk는 zero-copy를 지원하기 위하여 PF_RING을 이용하여 libzero 라이브러리를 구현하였다. libzero를 이용함으로써 패킷수집 스레드는 들어오는 패킷을 패킷처리 스레드로 메모리의 시간적 공간적 지역성을 높이기 위하여 배치형태로 패킷을 모아서 전달한다. 예를 들어 처음 32개의 패킷은 첫 번째 처리 스레드로 보내고 다음 32개의 패킷은 두 번째 처리 스레드로 보낼 수 있다. 최신 NIC들은 들어오는 패킷을 RSS를 사용하여 멀티큐에 하드웨어적으로 분산되도록 할지라도 n2disk에서는 이 기능을 사용하지 않았다. 왜냐하면, RSS는 패킷들이 네트워크 어댑터에서 수신되는 것과 다른 순서로 저장되도록 하기 때문이다. 패킷처리 스레드는 패킷들을 공유메모리로 복사해야 한다. 그리고 옵션으로 패킷들에 대하여 인덱스를 만들 수도 있다.



(그림 5) 멀티스레드 n2disk 시스템[3]

3. 시스템구성

n2disk는 2개의 시스템에서 테스트를 수행하였다.

- 싱글 프로세서 로우엔드 시스템(2.5GHz 쿼드코어와 Hyper Threading Xeon X3440). 각 코어는 약 168 CPU 사이클/패킷의 성능을 가진다. 이 숫자는 CPU 클락을 10기가 라인에서 초당 들어오는 패킷의 수로 나눈 것임(14.881 Mpps).
- Non Uniform Memory Access(NUMA) 하이엔드 시스템(듀얼 2.0GHz eight core와 HT Zeon E5-2650, ~134 CPU cycles/paclet). 이 시스템은 7개의 SSD를 사용하고 LSI 9260-16i 컨트롤러를 사용한다. 그리고 8개의 10k RPM 디스크를 사용하여 10기가 라인의 트래픽을 저장할 수 있음.

Ⅳ. 실시간 인덱스 생성

우리는 앞의 10Gbps 트래픽 저장기법을 사용하여서 10Gbps 트래픽을 저장할 수 있다. 그러나 효율적인 검색을 위하여 인덱스가 필요하다. 예를 들어서 우리는 다음과 같은 트래픽 데이터 검색을 위하여 인덱스를 필요로 한다.

Q1: SELECT COUNT(*), SUM(PKTS), SUM (BYTES) FROM NETFLOW

Q2: SELECT COUNT(*) FROM NETFLOW WHERE L4_SRC_PORT=80 OR L4_DST_PORT=80

Q3: SELECT COUNT(*) FROM NETFLOW GROUP BY IPV4_SRC_ADDR

Q4: SELECT IPV4_SRC_ADDR, SUM(PKTS), SUM(BYTES) AS s FROM NETFLOW GROUP BY IPV4_SRC_ADDR ORDER BY s DESC

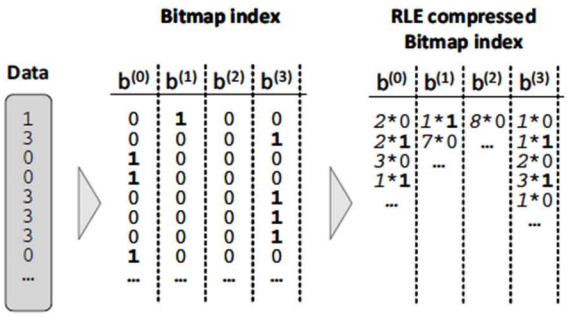
Q5: SELECT IPV4_SRC_ADDR, L4_SRC_PORT, IPV4_DST_ADDR, L4_DST_PORT, PROTOCOL, COUNT(*), SUM(PKTS), SUM(BYTES) FROM NETFLOW WHERE L4_SRC_PORT=80 OR L4_DST_PORT=80 GROUP BY IPV4_SRC_ADDR, L4_SRC_PORT, IPV4_DST_ADDR, L4_DST_PORT, PROTOCOL

일반적으로 인덱스를 이용하기 위하여 MySQL과 같은 데이터베이스를 사용할 수 있다. 그러나 현재 존재하는 데이터베이스의 쓰기성능이 10Gbps의 네트워크 속도보다 제한되어 있기 때문에 데이터베이스를 이용하여 10Gbps에 해당하는 네트워크 트래픽을 위한 인덱스를 만드는 것은 불가능하다[12].

한편으로 우리는 실시간으로 인덱스를 생성하는 것이 아니라 패킷이 저장된 이후에 트래픽 수집장치가 한가한 시간을 사용하여 사후처리로 인덱스를 생성할 수 있다. 예를 들어 하루에 한 번 심야의 시간에 인덱스를 생성할 수 있다. 이렇게 할 경우 트래픽 수집장치의 오버헤드를 줄일 수 있지만 아직 인덱스가 생성되지 않은 트래픽에 대한 검색을 하기 어려운 문제점이 존재한다[3]. 따라서 우리는 실시간 인덱스 생성방법에 초점을 맞추도록 한다.

우리는 실시간 인덱스를 생성하기 위하여 B-tree를 사용하는 것이 아니라 FastBit[13]과 같은 Bitmap Index 기법을 사용할 수 있다.

비트맵 인덱스는 검색을 빠르게 하는 구조이다[14]. 그것은 값들의 연속 집합을 바이너리 배열의 위치로 매핑한다. 예를 들어 (그림 6)은 0에서 3의 범위를 가지는 연속 데이터의 예를 보여준다. 예를 들어 두 번째 데이터 3의 존재는 두 번째 행의 마지막 컬럼을 1로 세팅함으로써 알 수 있다. 네트워크 애플리케이션에서 port 번호 데이터(n)를 가지는 m개의 레코드를 매핑하려고 할 때 요구되는 저장공간은 m*n 비트가 된다. 이러한 표현방식은 비트맵 인덱스 간의 bitwise 연산을 통하여 효율적인 검색을 가능하게 한다. 예를 들어 특정 소스 포트와 목적지 포트를 사용하는 모든 레코드를 두 개의 비트맵 인덱스의 bitwise 연산을 통하여 발견할 수 있다.



(그림 6) Bitmap Index and RLE compressed Bitmap Index[14]

비트맵인덱스의 단점은 많은 저장공간이 있어야 한다는 것이다. 이 문제를 해결하기 위하여 압축된 비트맵 인덱스가 제안된다. 이것은 Run Length Encoding (RLE)을 사용하여 컬럼방향으로 인덱스를 압축하는 것이다. (그림 6)에서 보는 것과 같이 연속된 1이나 0의 값들은 하나의 1 또는 0과 그것들의 개수로 표현될 수 있다. 이러한 방법으로 디스크 저장공간을 줄일 수 있다. 최근의 연구는 압축된 비트맵 인덱스가 많은 항목을 가질 경우에도 전형적인 B-트리보다 더 컴팩트하다는 것을 보여준다[15].

1. BBC

높은 압축률을 보여줄 뿐만 아니라 빠른 bit-wise 연산을 가능하게 하는 방법들이 있다. 그 중에 가장 잘 알려진 것은 Byte-aligned Bitmap Code(BBC)이다[16]-

[18]. BBC는 효율적으로 bit-wise 연산을 할 뿐만 아니라 gzip과 같은 정도로 압축할 수 있다.

BBC는 fil 이라고 불리는 연속적인 값들을 하나의 값과 그것들의 수로 표현하는 run-length encoding에 기반을 둔다. fill 비트가 0이면 0-fill이라고 부르고 1이면 1-fill이라 한다.

RLE 기법들은 fill의 길이와 short fill의 표현에 차이가 존재한다. 단순한 방법은 short-fill을 위하여 하나의 워드를 사용할 수 있는데 이것은 더 많은 공간을 사용하기 때문에 비효율적이다. 일반적인 개선책은 short fill을 따로 만드는 것이다. 그리고 두 번째 개선책은 fill 길이를 표현하기 위해 가능한 한 적은 비트를 사용하는 것이다.

비트 시퀀스가 주어졌을 때 BBC는 먼저 그것을 바이트로 나누고 그다음 바이트들을 run으로 그룹화한다. 각 BBC run은 tail이 따라오는 하나의 fill로 구성된다. BBC는 fill length를 비트 수가 아니라 바이트 수로 표현한다.

또 다른 특징은 byte alignment이다. 이것은 fill length가 바이트의 정수배가 되도록 한다. 왜냐하면, 각각의 비트에서 작업하는 것은 바이트에서 작업을 하는 것보다 대부분의 CPU에서 훨씬 더 비효율적이기 때문이다. alighment를 하지 않는 것은 압축률을 더 높게 할 수 있지만 bit-wise 연산의 속도를 느리게 한다.

2. WAH

최신 컴퓨터들은 보통 하나의 바이트에 접근하는데 하나의 워드에 접근하는 것과 같은 시간이 걸린다[19]. 이것을 이용하고 논리 연산 시간을 최소화하기 위하여 Word-aligned hybrid scheme(WAH)가 제안되었다. 주요 아이디어는 단지 2종류의 워드가 있도록 코딩 기법을 단순화하는 것이고 논리 연산 중에 개별 비트나 바이트를 추출할 필요가 없도록 정렬기법을 설계하는 것이다.

WAH는 RLE와 literal scheme의 하이브리드라는 점에서 BBC와 유사하지만, BBC와 달리 훨씬 더 간단하고 바이트가 아니라 워드로 압축된 데이터를 저장한다.

128 bits	1,20*0,3*1,79*0,25*1			
31-bit groups	1,20*0,3*1,7*0	62*0	10*0,21*1	4*1
groups in hex	40000380	00000000 00000000	001FFFFF	0000000F
WAH (hex)	40000380	80000002	001FFFFF	0000000F

(그림 7) A WAH bit vector[20]

WAH의 두 종류의 워드는 literal 워드와 fill 워드이다. WAH는 literal 워드 (0)과 fill 워드 (1)를 구별하기 위하여 워드의 가장 중요한 비트를 사용한다. literal 워드의 나머지 비트들은 비트맵으로부터의 비트 값들을 사용한다. 그리고 fill 워드의 두 번째로 중요한 비트는 fill 비트이고 나머지는 fill length를 표현한다.

(그림 7)은 128 비트를 표현하는 WAH 비트 벡터를 보여준다. 이 예에서 워드는 32비트이다. (그림 7)의 두 번째 라인은 비트맵이 어떻게 31비트 그룹으로 나누어지는지를 보여준다. 그리고 세 번째 라인은 각 그룹의 16진수 값을 보여준다. 마지막 라인은 WAH 워드의 값들을 보여준다. 처음 3개의 워드는 일반적인 워드이다. 2개의 literal 워드와 1개의 fill 워드를 나타낸다. fill 워드 80000002는 두 개의 워드 길이의 0-fill을 나타낸다. 즉, 62개의 연속된 zero 비트를 나타낸다. 다른 말로 하면 우리는 fill length를 literal 워드의 배수로 나타낼 수 있다.

(그림 8)은 Bitwise 연산의 예를 보여준다. 논리 연산을 수행하기 위하여 우리는 각 31비트 그룹을 매칭할 필요가 있다. literal 워드는 그 그룹의 위치를 차지하고 fill 워드는 해당하는 첫 번째 그룹을 위해 예약된 공간에 주어진다. C의 첫 번째 31비트 그룹은 A의 것과 같다. B의 대응하는 그룹이 1-fill이기 때문이다. C의 다음 3개의 그룹은 오직 zero 비트만을 포함한다.

논리 연산은 압축된 비트맵에서 직접적으로 수행될 수 있다. 연산에 필요한 시간은 압축된 비트맵의 크기에 관련된다. 압축된 부분과 압축되지 않은 부분의 비율이 0.5보다 작을 때 논리 연산 시간은 평균 압축률에 비례한다.

A	40000380	80000002		001FFFFF	0000000F
B	C0000002		7C0001E0	3FE00000	00000003
C	40000380	80000003			00000003

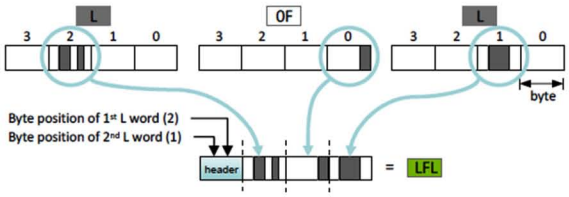
(그림 8) Bitwise logical AND operation, C = A AND B[20]

WAH는 BBC에 비해 훨씬 더 효율적이다. 첫째, WAH의 인코딩 방법은 훨씬 더 간단하다. WAH는 두 종류의 워드를 가지고 있고 BBC는 네 종류의 run을 가지고 있다. 둘째, WAH는 항상 워드를 액세스하지만 BBC는 바이트를 액세스한다. BBC는 메모리로부터 CPU 레지스터로 데이터를 로드하는 데 더 많은 시간이 걸린다. 셋째, BBC는 WAH보다 더 작은 short-fill을 인코드할 수 있다. 그러나 그것은 더 많은 비용이 든다.

3. COMPAX

COMPressed Adaptive index(COMPAX)[21]는 비트맵 인덱스 사이즈를 줄이는 워드의 코드북을 사용하여 생성된다. 네 개의 워드 타입이 사용된다. 다음의 네 개의 32비트 워드 타입들은 31비트 청크 단위로 비트 스트림을 인코드하는 데 사용된다.

1) [L]은 0과 1의 bitwise mix를 구성하는 청크를 나타낸다.
   1 0011100 11000010 00110000 00000000
2) 0-fill [0F]는 RLE에 의해 제로비트의 연속적인 청크들을 인코드한다. 예를 들어 3개의 31 제로비트가 단 하나의 32비트 워드로 다음과 같이 표현된다.
   000 00000 00000000 00000000 00000011
   처음 3개의 비트(000)은 코드워드 타입을 나타내고 나머지 29비트는 31비트 청크의 수를 나타낸다.
3) [LFL]은 null-suppression을 적용한 후에 [L]-[0F]-[L] 워드의 시퀀스를 나타낸다. 세 개의 워드 중에서 페이로드 바이트 중 오직 하나만 non-zero ("dirty") 이고 [0F] 워드에서 더티 바이트가 포지션 0에 있으면 [L] 워드로부터의 세 개의 더티 바이트와 두개의 위치 (0에서 3)는 단 하나의 [LFL] 워드로 표현될 수 있다. (그림 9)는 이것의 예를 보여준다.
4) [FLF] 워드는 [0F]-[L]-[0F]를 따르는 워드의 시퀀스를 나타낸다. 세 개의 연속된 워드가 그 패턴을 따르고 각 페이로드에 단 하나의 더티 바이트가 있고 [0F] 워드에서 더티 바이트가 포지션 0에 있을 때 그것들은 [FLF]로 압축된다.



(그림 9) [L]-[F]-[L]로부터 [LFL] 코드워드 생성[21]

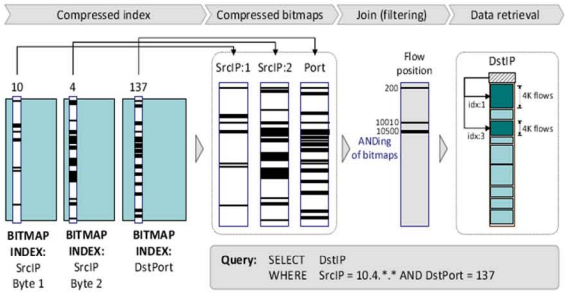
WAH와 비교할 때 WAH는 두 개의 워드를 사용한다. COMPAX는 [LFL] 과 [FLF]의 코드워드를 사용한다. 그리고 COMPAX는 1-fil을 사용하지 않는다. COMPAX는 WAH에 비교하여 저장공간을 훨씬 더 적게 사용한다. 튜플 정렬단계와 함께 사용할 때 약 60% 이상의 공간을 절약할 수 있었다. 예를 들어 8.1기가바이트 WAH 비트맵 인덱스는 COMPAX를 가지고 3.3기가바이트로 줄어들었다.

4. 질의 처리 방법

시스템은 소스 IP, 목적지 IP, 소스포트, 목적지포트, 프로토콜, 시간과 같은 인덱스에 포함되는 속성에 대하여 equality 질의나 범위 질의를 수행할 수 있다. 질의 수행은 (그림 10)과 같이 다음의 단계를 포함한다.

1) 질의에 포함된 컬럼이나 속성이 결정된다. 압축된 비트맵 인덱스로부터 관련된 컬럼이 검색된다.
2) 압축된 컬럼들 사이의 boolean 연산은 decompression 없이 직접 수행된다. 압축된 파일에서 플로우 레코드 위치가 결정된다.
3) 파일의 해당 부분의 압축을 풀고 결과가 사용자에게 전달된다.

예를 들어 시스템 관리자는 10.4.0.0/16 범위에 있는



(그림 10) 질의처리방법 예제[21]

노드에 의해 포트 137을 접속한 모든 목적지 아이피를 알기 원한다고 하자. 해당 질의는 다음과 같다.

- Query: 소스IP가 10.4.*.* 범위에 있고 목적지 포트가 137인 모든 목적지 IP를 찾아라.

비트맵 인덱스 파일들은 매시간 만들어지기 때문에 첫째 질의의 관심 시간 범위에 있는 인덱스 파일들이 검색된다. 그리고 SrcIP.byte1, SrcIP.byte2, DstPort 에 대한 비트맵 인덱스들이 검색된다. 우리는 각 컬럼의 압축을 풀 필요가 없다. 그리고 세 개의 컬럼에 대하여 AND 연산이 수행된다.

세 컬럼의 조인 결과가 {200, 10010, 10500}의 플로우 번호에 해당한다고 가정하자. 각 압축블록은 4000개의 플로우 레코드를 포함한다. 그러므로 해당하는 3개의 플로우를 검색하기 위해서 우리는 첫 번째 블록과 세 번째 블록을 검색할 필요가 있다. 각 블록의 시작 위치는 해당 파일의 헤더에 제공된다. 그리고 마지막으로 해당하는 소스 IP의 주소가 사용자에게 전달된다.

V. 결론

우리는 이 문건에서 10Gbps 네트워크 트래픽 저장방법과 실시간 인덱스 생성방법에 대하여 소개하였다. 네트워크 트래픽 저장을 위하여 PF_RING 기술을 소개하였고 싱글 스레드 시스템과 멀티 스레드 시스템에서 네트워크 트래픽을 저장하는 방법을 소개하였다.

그리고 저장된 네트워크 트래픽을 효율적으로 검색하기 위한 비트맵 인덱스 기법을 소개하였다. 그리고 비트맵 인덱스 기법으로 BBC, WAH, COMPAX를 소개하였고 최종적으로 비트맵 인덱스를 이용한 질의처리방법에 대하여 소개하였다.

용어해설

플로우데이터 플로우데이터는 크게 5투플(소스아이피, 소스포트, 목적지아이피, 목적지포트, 프로토콜)과 시작시간, 종료시간으로 구성된다. 5투플이 같은 여러 개의 패킷이 존재할 수 있는데 일정시간 동안에 포함되는 여러개의 패킷들을 하나의 플로우로 간주함.

비트맵인덱스 대용량 네트워크 데이터를 검색하기 위해서는 인덱스가 필요하다. 일반적인 데이터베이스로는 인덱스를 지원하기에 어려울 수 있으므로 비트맵인덱스를 사용한다. 비트맵인덱스는 Bit-wise 연산을 이용하여 다양한 질의에 대한 처리를 효율적으로 수행할 수 있음.

약어 정리

IDS	Intrusion Detection System
IPS	Intrusion Prevention System
BPF	BerkeleyPacket Filter
RSS	Receive-Side Scaling
NIC	Network Interface Card
NUMA	Non Uniform Memory Access
RLE	Run Length Encoding
BBC	Byte-aligned Bitmap Code
WAH	Word-aligned hybrid scheme
COMPAX	COMPressed Adaptive indeX

참고문헌

[1] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *Winter USENIX Conf.*, 1993.

[2] S. McCanne, C. Leres, and V. Jacobson, "Libpcap," Lawrence Berkeley National Labs, 1994, http://www.tcpdump.org

[3] L. Deri, A. Cardigliano, and F. Fusco, "10 Gbit Line Rate Packet-to-Disk Using n2disk," *TMA*, 2013.

[4] F. Fusco and L. Deri, "High Speed Network Traffic Analysis with Commodity Multi-Core Systems," *IMC*, 2010.

[5] Intel White Paper, "Accelerating High-Speed Networking with Intel I/O Accelerating Technology," 2006.

[6] Intel White Paper, "Intelligent Queing Technologies for Virtualization," 2008.

[7] L. Deri et al., "Wire-Speed Hardware-Assisted Traffic Filtering with Mainstream Network Adapters," NEMA, 2010.

[8] L. Deri et al., "Improving Passive Packet Capture: Beyond Device Polling," *SANE*, 2004.

[9] A. Cox, "Network Buffers and Memory Management," *Linux J.*, no. 30, Oct. 1st, 1996.

[10] L. Rizzo, "Device Polling Support for FreeBSD," BSDConEurope Conf., 2001.

[11] R. Love, "CPU Affinity," *Linux* J., no. 111, Jul. 1st, 2003.

[12] L. Deri, V. Lorenzetti, and S. Mortimer, "Collection and Exploration of Large Data Monitoring Sets Using Bitmap Databases," *TMA, LNCS*, vol. 6003, 2010, pp. 73-86.

[13] K. Wu et al., "FastBit: Interactively Searching Massive Data," J. Phys: Conf. Ser., SciDAC, vol. 180, no. 1, 2009.

[14] F. Fusco et al., "PcapIndex: An Index for Network Packet Traces with Legacy Compatibility," *ACM SIGCOMM Computer Commun. Rev.*, vol. 42, no. 1, Jan. 2012, pp. 47-53.

[15] K. Wu, E. Otoo, and A. Shoshani, "On the Performance of Bitmap Indices for High Cardinality Attributes," *VLDB Conf.*, 2004.

[16] G. Antoshenkov, "Byte-alighed Bitmap Compression," *Proc. DCC*, 1995.

[17] G. Antoshenkov and M. Ziauddin, "Query Processing and Optimization in ORACLE Rdb," *VLDB J.*, vol. 5, no. 4, Dec. 1996, pp. 229-237.

[18] T. JohnSon, "Performance Measurements of Compressed Bitmap Indices," *VLDB*, 1999, pp. 278-289.

[19] D.A. Patterson, J.L. Hennessy, and D. Goldbert, "Computer Architecture: A Quantative Approach," Morgan Kaufmann, 2nd edition, 1996.

[20] K. Wu, E.J. Otoo, and A. Shoshani, "Compressing Bitmap Indexes for Faster Search Operations," *IEEE SSDBM*, 2002.

[21] F. Fusco, M.P. Stoecklin, and M. Vlachos, "NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic," *Proc. VLDB Endowment*, vol. 3, no. 2, 2010.