논문 15-40-09-15 The Journal of Korean Institute of Communications and Information Sciences '15-09 Vol.40 No.09
http://dx.doi.org/10.7840/kics.2015.40.9.1754

고속 패킷 분류를 위한 2차원 비트맵 트라이

서 지 희◆, 임 혜 숙°

2-Dimensional Bitmap Tries for Fast Packet Classification

Ji-hee Seo◆, Hye-sook Lim°

요 약

인터넷 라우터에서 행해지는 패킷분류는 5가지 패킷 헤더를 검사하여 선속도로 처리해야하기 때문에, 라우터가 수행하기 어려운 기능 중 하나이다. 본 논문에서는 리프-푸싱 된 영역분할 사분트라이 기반 비트맵 트라이 구조(leaf-pushed AQT bitmap trie)를 제안한다. 제안하는 구조는 영역분할 사분트라이(Area-based Quad Trie, AQT)에 기초하되 불필요한 칩-외부메모리 접근을 줄이고자 리프-푸싱(leaf-pushing)기법을 적용하고, 멀티 비트 트라이인 비트맵을 적용하여 패킷분류 속도와 확장성을 모두 향상시킨 구조이다. 성능 평가를 위하여 ACL FW, IPC 룰 셋을 각각 1k, 5k, 10k로 크기를 증가시키며 실험을 진행하였다. 그 결과, 제안하는 구조에서는 룰 셋의 종류나 크기와는 상관없이 패킷분류를 위하여 필요한 칩-외부메모리 접근 횟수가 1번 이내임을 확인할 수 있었다. 또한, 비트맵 트라이를 적용함으로 인해, 리프-푸싱기반 영역분할 사분트라이와 비교하여 약 50%의 칩-내부메모리 접근이 요구됨을 보았으며, 또한 칩-내부메모리 요구량의 변화폭이 룰 셋이 증가함에 따라 안정적으로 변화하여 제안하는 구조의 확장성을 확인할 수 있었다.

Key Words : Bitmap, Leaf-pushing, AQT, Packet Classification

ABSTRACT

Packet classification carried out in Internet routers is one of the challenging tasks, because it has to be performed at wire-speed using five header fields at the same time. In this paper, we propose a leaf-pushed AQT bitmap trie. The proposed architecture applies the leaf-pushing to an area-based quad-trie (AQT) to reduce unnecessary off-chip memory accesses. The proposed architecture also applies a bitmap trie, which is a kind of multi-bit tries, to improve search performance and scalability. For performance evaluation, simulations are conducted by using rule sets ACL, FW, and IPC, with the sizes of 1k, 5k, and 10k. Simulation results show that the number of off-chip memory accesses is less than one regardless of set types or set sizes. Additionally, since the proposed architecture applies a bitmap trie, the required number of on-chip memory accesses is the 50% of the leaf-pushed AQT trie. In addition, our proposed architecture shows good scalability in the required on-chip memory size, where the scalability is identified by the stable change in the required memory sizes, as the size of rule sets increases.

Ⅰ. 서 론

오늘날 인터넷은 우리 생활의 많은 다양한 분야에서 사용되고 있다. 인터넷 사용자의 수는 날이 갈수록 기하급수적으로 증가하는 추세이며, 라우터에서 처리해야하는 데이터인 '패킷(packet)'의 종류와 수 또한

※ 본 연구는 한국연구재단(NRF)의 중견연구자지원사업 핵심과제(2014R1A2A1A11051762)와 정보통신기술진흥센터의 대학 ICT연구센터 육성지원사업((IITP-2015-H8501-15-1007)의 연구비 지원으로 수행하였습니다.
◆ First Author : Ewha Womans University Department of Electronics Engineering, jjjhseo@naver.com, 학생회원
° Corresponding Author : Ewha Womans University Department of Electronics Engineering, hlim@ewha.ac.kr 종신회원
논문번호 : KICS2015-07-207, Received July 2, 2015; Revised September 3, 2015; Accepted September 3, 2015

빠른 속도로 증가하고 있다. 현재, 인터넷은 응용프로그램의 특성에 따라 특정 QoS(Quality of Service)정책[1]으로 데이터를 처리하거나, 외부 망으로부터 내부망을 보호하기 위해 방화벽(firewall) 서비스, 트래픽 사용량을 분배시키기 위한 로드 밸런싱(load balancing)과 같은 다양한 형태의 서비스를 제공하는 데에 초점을 맞추고 있다.[2] 인터넷이 사용자에게 이와 같은 서비스를 제공하기 위해서 라우터에서는 패킷을 클래스에 따라 분류하는 패킷분류(Packet classification) 과정이 필수적으로 수행되어야 한다.[3]

패킷분류는 패킷의 헤더에 있는 많은 필드(field) 중에서 5개의 필드를 고려하여 이루어지는데, 이때 고려되어지는 5가지 필드는 목적지 IP 주소, 근원지 IP 주소, 목적지 포트 번호, 근원지 포트 번호, 프로토콜 정보이다.[4] 라우터에서는 검색(search) 과정을 통해 인터넷 사용자에 의해 생성된 입력 패킷과 일치하는 룰(rule)들을 모두 찾은 뒤, 이 중에서 최우선순위를 갖는 룰의 클래스에 따라 입력 패킷을 처리하는 방식으로 패킷분류가 수행된다. 목적지 IP 주소와 근원지 IP 주소에 대해서는 프리픽스 일치(prefix matching), 목적지 포트 번호와 근원지 포트 번호에 대해서는 영역 일치(range matching), 프로토콜 정보에 대해서는 완전 일치(perfect matching) 여부를 확인한다.[5] 패킷이 기가 비트(Giga bit) 또는 테라 비트(Tera bit)속도로 빠르게 라우터에 도달하기 때문에, 라우터에 들어온 패킷은 약 나노세컨드(nanoseconds) 정도의 짧은 시간동안만 라우터에 머물 수 있으며, 짧은 시간 안에 처리되어 포워딩(forwarding)되어야 하므로, 라우터의 패킷 처리 속도는 라우터의 성능을 평가하는 데에 있어서 중요한 요소로 작용한다. 하지만, 패킷 포워딩 알고리즘을 이용하여 기가 비트 또는 테라 비트 속도의 패킷 포워딩을 구현하는 것은 불가능하기 때문에, 대부분의 패킷 포워딩 엔진은 같은 기능을 수행하는 블럭(block)들을 병렬적으로 연결하거나, 파이프라이닝(pipelining)을 이용한다.

라우터의 패킷 처리 속도를 향상시키고자 한 연구는 이전에도 존재하였는데, 이는 크게 빠른 메모리 접근 속도를 갖는 별도의 하드웨어를 사용하는 하드웨어 기반 연구, 새로운 프로토콜을 사용하는 프로토콜 기반 연구, 포워딩 테이블의 자료구조를 개선하는 소프트웨어 기반의 연구로 나눌 수 있다.[6] 본 연구에서는 라우터 성능 향상에 있어서의 패킷 처리속도 문제를 새로운 포워딩 테이블의 구조와 검색 알고리즘을 이용하여 해결하고자 하였다.

소프트웨어 측면에서 라우터에서의 패킷처리 속도, 즉 검색 속도를 향상시키기 위해 패킷의 여러 필드를 동시에 검색하는 알고리즘들이 연구되어 왔으며, 5가지 헤더필드 중에서 플로우(flow)의 다양성이 큰 근원지 IP주소와 목적지 IP주소를 이용한 다차원 검색이 효율적이라는 연구결과가 보고된 바 있다.[7] 또한, 칩-내부메모리(on-chip memory)는 주 메모리를 사용하는 반면, 칩-외부메모리(off-chip memory)는 보조 기억장치를 사용하기 때문에, 칩-내부메모리에 접근하는 것에 비해 오랜 시간이 걸린다.[9][10] 또한, 검색 과정에 있어서, 칩-내부메모리에 접근 하는 횟수가 칩-외부메모리에 접근하는 횟수에 비해 상대적으로 많기 때문에, 칩-외부메모리 접근 횟수 뿐 아니라 칩-내부메모리 접근 횟수를 모두 고려한 구조를 설계할 필요가 있다. 이러한 기존 연구결과들을 바탕으로, 본 논문에서는 다차원 검색을 위해 근원지 IP 프리픽스와 목적지 IP 프리픽스를 조합한 이진트라이인 영역분할 사분 트라이(Area-based Quad Trie, AQT)[8]를 이용하되, AQT에 리프-푸싱과 비트맵구조를 적용하여, 검색속도를 향상시킨 구조를 제안한다.

Ⅱ. 관련 연구

2.1 영역분할 사분 트라이[11]

AQT는 패킷 헤더 중 플로우의 다양성이 가장 큰 근원지 IP프리픽스 필드와 목적지 IP프리픽스를 영역으로 취급하여 패킷분류를 수행하는 기본적인 영역분할 알고리즘이다. 근원지 IP프리픽스 필드를 x축으로 하고, 목적지 IP 프리픽스 필드를 y축으로 갖는 정사각형 평면을 전체 검색 영역으로 취급한다. W를 최대 프리픽스 길이라 한다면, IPv4에서의 W는 32를 의미하고, IPv6에서의 W는 128을 의미한다. AQT의 전체 검색 영역은 x축의 길이가 2^W이고 y축의 길이가 2^W인, 넓이가 2^W*2^W인 정사각형에 해당한다. 전체 검색영역 중에서 근원지 프리픽스 길이가 i, 목적지 프리픽스 길이가 j인 룰에 대해서는 $2^{(W-i)}*2^{(W-j)}$만큼의 넓이를 차지하는 사각형으로 표현되기 때문에, 프리픽스의 길이가 길수록 차지하는 영역의 넓이는 좁아진다.

AQT의 구축(Build)과정에서 근원지 프리픽스에 해당하는 x축에 대해서 주어진 영역을 절반으로 나누었을 때, 왼쪽영역에는 0을, 오른쪽영역에는 1을 할당하며, 목적지 프리픽스에 해당하는 y축에 대해서는 주어진 영역을 절반으로 나누었을 때, 위쪽영역에는 1,

아래쪽영역에는 0을 할당한다. 예를 들어, 근원지 프리픽스가 0이고, 목적지 프리픽스가 10인 0번 룰(*R*0)을 구축할 경우, 0번 룰에 대한 영역은 그림1의 왼쪽 그림과 같다. 또한 AQT에서 룰을 구축하는 과정에서는 근원지 프리픽스와 목적지 프리픽스를 1비트씩 조합하여 코드워드(codeword)를 생성하며, 코드워드의 길이는 근원지 프리픽스와 목적지 프리픽스 중 짧은 길이의 두 배와 같다. 0번 룰의 경우, 근원지 프리픽스가 0으로 길이가 1비트이고, 목적지 프리픽스가 10으로 길이가 2비트이므로 코드워드의 길이는 2비트가 된다, 이 경우, 근원지 프리픽스와 목적지 프리픽스를 각각 1비트씩 조합해보면 01이므로, AQT에 나타내면 [그림 1]의 오른쪽그림과 같다.

각각의 코드워드를 갖는 분할된 영역에서 입력 패킷과 일치하는 룰을 검색하기 위해서는 각 영역에서 크로싱 필터 셋(Crossing Filter Set, CFS)을 정의해야 한다. 룰이 차지하는 영역이 선택된 영역의 두 필드 중 어느 한 필드 부분이라도 모두 차지하고 있다면, 그 룰은 선택된 영역의 크로싱 필터 셋에 포함되는 것으로 정의된다. 이러한 이유로 하나의 크로싱 필터 셋에 여러 개의 룰이 포함되어 있을 수 있다. 입력 패킷의 경우 근원지 프리픽스와 목적지 프리픽스가 각각 최대 프리픽스 길이인 *W*로 주어지므로, 입력패킷은 검색영역 상에서 하나의 점으로 표현되고, 점이 포함되는 영역에 해당하는 룰들에 대해, IP주소 정보에 대해서는 프리픽스일치, 포트번호에 대해서 영역일치, 프로토콜정보에 대해서는 완전일치 여부를 확인하여, 모두 일치하는 룰 중 우선순위가 가장 높은 룰을 검색결과로 반환한다.

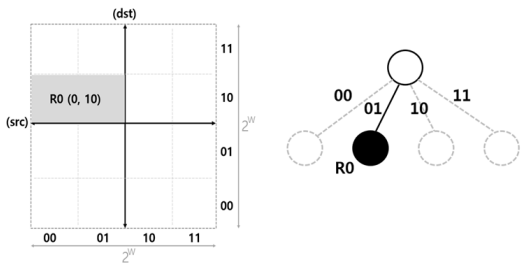

그림 1. 근원지 IP프리픽스 0, 목적지 IP프리픽스 10인 0번 룰에 대한 영역(좌)과 AQT(우)로 표현한 예
Fig. 1. The example of 'range of rule'(left) and 'AQT'(right) in case of source IP prefix '0*' and destination IP prefix '10*'

2.2 리프-푸싱(leaf-pushing)

리프-푸싱이란 프리픽스 노드(prefix node)가 트라이의 리프가 아닌 내부노드에 있을 경우, 리프로 이동시켜, 해당 트라이에서 모든 프리픽스 노드가 리프노드에만 존재하도록 만드는 것이다. 다시 말해, 리프-푸싱을 통하여 프리픽스 노드들은 서로 포함관계가 없는 디스조인트(disjoint)한 관계가 된다. 또한, 트라이가 리프-푸싱 될 경우, 프리픽스 노드가 리프노드에만 존재하기 때문에, 검색(search)시 내부 노드에 접근한 경우, 접근한 레벨보다 상위레벨에는 프리픽스 노드가 없다는 것이 보장되기 때문에, 불필요한 검색과정을 줄일 수 있는 장점이 있다.

트라이에 존재하는 내부 프리픽스 노드에 대하여 한 레벨씩 아래로 푸싱을 진행하되, 리프가 되거나, 더 긴 프리픽스를 만나는 경우 푸싱을 종료한다. 예를 들어, [그림 2]에서 내부노드 프리픽스 P1(001*)이 리프-푸싱되는 과정은 다음과 같다. 왼쪽 자식노드에는 더 긴 리프 프리픽스인 P3(0010*)가 존재하므로, 리프-푸싱되지 않고, 오른쪽에는 노드가 존재하지 않으므로 리프-푸싱되는데, 리프-푸싱된 0011*는 리프노드이므로 P1에 대한 리프-푸싱이 종료된다. 내부노드인 P4(0*)의 경우에도, 오른쪽 자식노드로 리프-푸싱이 되어 생성된 노드는 리프노드이므로, P4는 '01' 노드에 저장된다. 하지만, 왼쪽 자식노드인 '00'은 리프노드가 아니므로 한 레벨 아래로 리프-푸싱이 일어나야 하지만, '00' 노드의 자식노드인 '000'과 '001'에는 이미 프리픽스가 존재하므로, P4는 더 이상 리프-푸싱되지 않는다. 그 결과, [그림 2]의 왼쪽 트라이의 리프-푸싱된 결과는 [그림 2]의 오른쪽 그림과 같다.

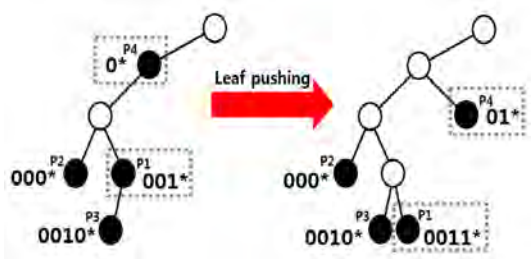

그림 2. 이진트라이의 리프-푸싱 예시
Fig. 2. The example of Leaf-pushing in a Binary Trie

2.3 비트맵 트라이(Bitmap Trie)[12]

비트맵 트라이는 멀티비트 트라이(multi-bit trie)의 한 종류로, 입력 데이터의 한 비트씩 검색을 진행하는 단순 이진트라이와는 달리, 입력 데이터의 여러 비트를 동시에 사용하여 검색을 수행하기 때문에 검색 속도가 빠르다. 또한, 트라이를 업데이트하는 데에 있어

서, 전체 트라이 크기를 고려하는 것이 아니라 트라이의 일부인 서브트라이의 크기에 한정되기 때문에, 업데이트 속도가 빠르다는 장점이 있다. 단순 이진트라이 구조를 이루는 기본 구성성분이 노드(node)였다면, 비트맵트라이를 이루는 기본 구성성분은 서브트라이(subtrie)이다. 기능적인 측면에서 본다면, 서브트라이는 크게 내부비트맵(internal bitmap)과 외부비트맵(external bitmap)으로 나누어 표현될 수 있으며, 구축과정을 [표 1]과 [그림 3], [그림 4]를 사용하여 설명하면 다음과 같다.

내부비트맵은 접근한 인덱스(index)에 해당하는 노드가 프리픽스 노드인지의 여부를 의미한다. 하나의 서브트라이 안에 존재하는 stride 크기는 고정적인 값이 아니지만, 본 논문이 제안하는 구조를 내부메모리에 구현하기 위해 요구되는 메모리 량이 구현 가능한 범위에 있기 위한 stride 크기가 2이므로, 본 논문에서 제안하는 구조 및 언급되는 비트맵 트라이에 대해서 stride를 2로 하였다. 1차원 비트맵트라이의 경우, [표 1]을 참고하면 N_{child}가 2이고, stride는 2이므로, 하나의 서브트라이의 1번째 레벨에는 1개, 2번째 레벨에는 2개, 3번째 레벨에는 4개의 노드가 존재할 수 있으므로, 하나의 서브트라이 안에 최대로 존재할 수 있는 노드의 개수가 내부비트맵의 길이가 된다. 다시 말해, 내부비트맵의 길이는 1+2+4 = $(2^{(2+1)}-1)$비트가 된다. 서브트라이에 포함된 노드들에 대해 0부터 $(2^{(stride갯수+1)}-2)$ = 6까지 번호를 매기고, 노드가 프리픽스 노드라면 노드번호에 해당하는 서브트라이 인덱스에서의 내부비트맵 값을 1로하고, 나머지 인덱스에 대해서는 내부비트맵 값을 0으로 한다. [그림 3]에서 0번 서브트라이의 내부비트맵은 0000100이라 할 수 있다.

외부비트맵은 서브트라이의 마지막 레벨에 있는 노드가 자식노드를 갖는지에 대한 여부를 나타내며, 자식노드를 가질 경우 1을, 갖지 않을 경우 0을 할당한다. [그림 4]는 [그림 3]의 0번 서브트라이에 대해 외부비트맵을 표현한 예이며, [그림 4]를 고려해 볼 때, 접근한 서브트라이의 마지막 레벨에 존재할 수 있는 최대 노드의 개수는 4개($N_{child}^{(stride갯수)}$) 이며, 각각이 모두 2개($N_{\chi ld}$)의 자식노드를 가지므로, 외부 비트맵의 길이는 $N_{child}^{(stride갯수)} * N_{\chi ld}$

= $N_{child}^{(stride갯수+1)}$ = $2^{(2+1)}$= 8비트가 된다.

[그림 3]에서 서브트라이의 마지막 레벨에 있는 노드인 3, 4, 5, 6번 노드의 자식노드가 모두 있다는 가정 하에 3, 4, 5, 6번 노드의 모든 자식노드들에 대해 왼쪽에서부터 외부비트맵의 인덱스를 부여한다. 그 결과, [그림 4]의 0번 서브트라이의 외부 비트맵은 11000000이 된다. 구축(Build)된 프리픽스(P1, P2, P3, P4)가 [그림3]와 같고, 입력주소가 010이라고 가정하고 검색과정의 일부를 통해 내부비트맵과 외부비트맵이 이용되는 과정을 살펴보면 다음과 같다.

현재 서브트라이는 stride가 2이므로, 서브트라이에서 검색을 진행할 수 있는 비트 수는 2비트이고, 0번 서브트라이에서부터 검색을 진행한다. 010에 대해서 최상위 비트로부터 2비트인 01로 검색을 진행한다면, 레벨1에서는 1번, 레벨2에서는 4번 노드에 도달하며, 4번 노드에서의 내부비트맵은 인덱스4에 위치한 값으로 1이다. 이는 4번 노드에 프리픽스가 저장되어있음을 의미하므로, 해당노드에 저장된 노드인 P4를 기억하고 상위레벨로 검색을 진행한다. 상위레벨인 레벨1에서는 1번 노드에서의 내부비트맵은 1번 인덱스에

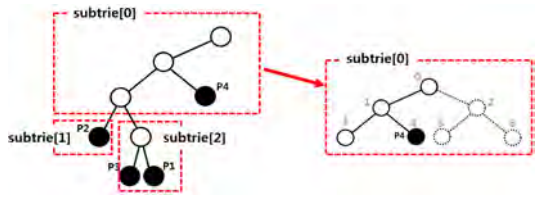

그림 3. 서브트라이 0번에 대한 내부 비트맵 예시
Fig. 3. The example of Internal Bitmap at subtrie[0]

표 1. 내부비트맵과 외부비트맵을 표현한 식
Table 1. The equation of internal bitmap and external bitmap

internal bitmap(bit)	$(N_{child}^{(stride갯수+1)}$-1) / ($N_{\chi ld}$ -1)
external bitmap(bit)	$N_{child}^{(stride갯수+1)}$

* $N_{\chi ld}$: 1개의 노드에서 가질 수 있는 최대 자식 노드의 수
* stride갯수 : 1개의 서브트라이 안에 존재하는 엣지(edge)또는 레벨(level)의 수

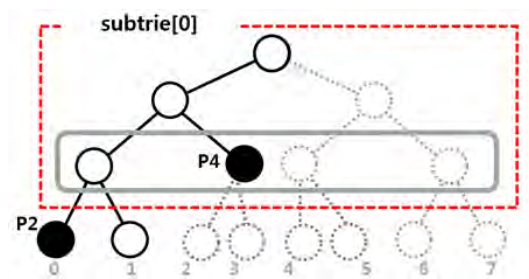

그림 4. 0번 서브트라이에서 외부비트맵을 나타내는 예시
Fig. 4. The example of 'External Bitmap' at subtrie[0]

해당한 값이며, 이 값이 0이므로, 1번 노드에는 프리픽스가 저장되어있지 않음을 의미한다. 이러한 과정을 해당 서브트라이의 루트노드에 도달할 때까지 반복하며, 루트 노드에 도달하게 되면 외부비트맵을 이용하여, 다음 서브트라이로의 진행여부를 결정한다. 입력 주소 010에 대하여 서브트라이에서 검색이 진행된 비트의 다음에 있는 1비트로 진행하였을 경우에 도달하는 노드는 외부비트맵의 인덱스 2에 해당하는 노드임을 [그림 4]를 통해 할 수 있다.

Ⅲ. 제안하는 구조

검색 속도를 향상시키기 위해서는 검색속도를 저하시키는 요인을 줄이거나, 검색속도를 향상시키는 요인을 추가해야 한다. 본 논문에서는 외부메모리 접근을 검색속도를 저하시키는 치명적인 요인으로 보았고, 이를 줄이기 위하여 영역분할 사분트라이(Area-based Quad Trie, AQT)를 사용하였다. 영역분할 사분트라이는 근원지 IP 프리픽스와 목적지 IP 프리픽스 정보를 사용하여, 2차원 트라이를 구성하기 때문에, 두 개가 아닌 하나의 트라이를 가지고 2개의 헤더필드 영역(근원지 주소 프리픽스, 목적지 주소 프리픽스)을 고려한 검색이 가능하며, 이로 인해 검색과정에서의 외부 메모리 접근을 줄일 수 있다.

반면 영역분할 사분트라이를 이용한 검색과정에서 룰 노드에 접근할 경우, 무조건적으로 외부메모리에 접근하여, 해당 노드에 저장된 룰들에 대해 룰 검색이 이루어지는 것에서 불필요한 메모리 접근이 이루어지고 있음을 볼 수 있었다. 이러한 불필요한 메모리 접근을 줄이기 위해서는 검색과정에서 접근하는 룰 노드의 수를 줄여야 한다고 여겼으며, 리프-푸싱 과정을 영역분할 사분트라이에 적용함으로서 검색과정에서의 룰 노드 접근을 1번 이하로 줄였다. 이는 곧, 1개의 패킷에 대한 검색과정에서 외부메모리에 접근이 최대 1번임을 의미한다. 또한, 제안하는 구조는 검색속도를 저하시키는 외부메모리 접근 횟수를 최소화하는 데에 그치지 않고, 입력 패킷에 대한 정보를 불러들이는 속도와 업데이트 속도를 검색속도를 위해 더 향상시켜야 할 요소로 여기고, 이를 향상시키기 위해 비트맵트라이를 적용하여, 최종적으로 리프-푸싱 된 영역분할 사분트라이를 기반으로 한 비트맵트라이(Leaf-pushed AQT Bitmap)를 제안한다.

3.1 자료구조(Data structure)

[그림 5]에서 보는 바와 같이, 내부메모리는 구축(Build)된 여러 개의 서브트라이와 하나의 Rule_pointer 배열로 이루어져 있다. 구축이 이루어진 서브트라이(subtrie)에는 Internal_bitmap, External_bitmap, Cumulate_subtire, Start_pointer로 이루어져 있으며, 각각의 내부메모리 요소들은 다음과 같은 기능을 수행한다.

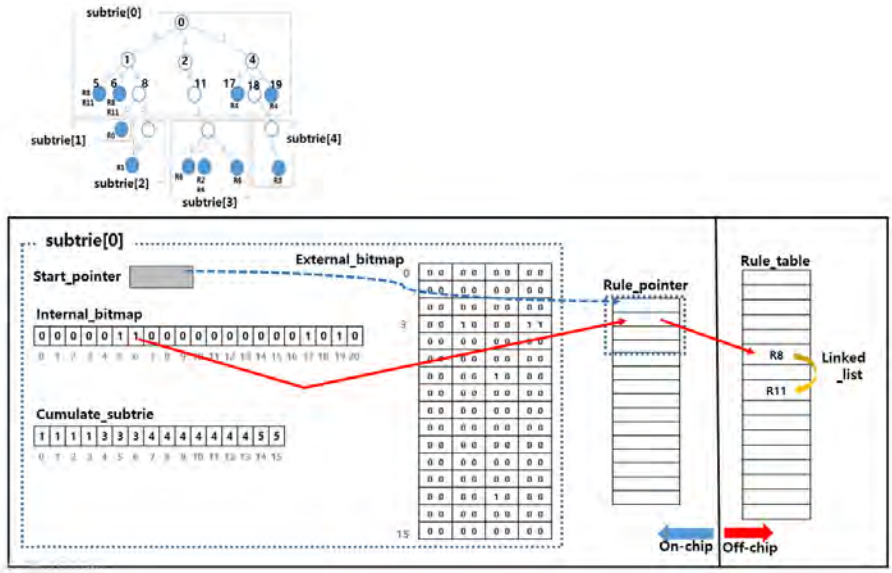

그림 5. 리프-푸싱 영역분할 사분트라이 기반 비트맵트라이의 메모리 구조
Fig. 5. The memory structure of 'Leaf-pushed AQT Bitmap trie'

- Internal_bitmap

비트맵 트라이에서의 '내부 비트맵'을 배열로 표현한 요소이다. 접근한 서브트라이에서 접근한 노드에 룰이 저장되어 있는 경우, 접근한 서브트라이에서의 내부비트맵을 나타내는 Internal_bitmap[subtrie번호] 배열에 대하여 접근한 노드의 노드번호에 해당하는 인덱스에 저장된 값을 1로 하고, 접근한 노드에 룰이 저장되어 있지 않은 경우에는 0으로 한다.

- External_bitmap

비트맵 트라이에서의 '외부 비트맵'을 2차원 배열로 표현한 요소이다. 행(row)번호는 하나의 서브트라이의 마지막 레벨에 존재할 수 있는 노드의 최대 개수와 같고, 각각의 행은 하나의 서브트라이의 마지막 레벨에 있는 노드가 마지막 레벨에서 몇 번째 노드인지를 의미하며, 1개의 행(row)에 존재하는 4개의 열(column)은 행 번호에 해당하는 노드에 대한 4개의 자식노드에 대한 정보를 나타낸다. 이때, 왼쪽에 있는 자식노드에서부터 가장 작은 열(c)번호의 열에 자식노드의 정보가 채워진다. 1개의 행(row)에 대한 1개의 열(column)에는 해당하는 행에 대응하는 노드의 자식노드가 존재하는 지에 대한 여부와 서브트라이의 마지막 레벨에 있는 노드 1개에 대해 해당노드가 몇 번째 노드인지에 대한 정보가 저장된다. 예를 들어, (1, 3)은 해당하는 서브트라이 마지막레벨의 노드의 자식노드가 존재하고, 이 자식노드가 마지막레벨의 노드의 3+1=4번째 자식노드임을 의미한다.

- Cumulate_subtrie

접근한 서브트라이의 마지막 레벨에서 모든 노드가 존재한다고 간주하고, 가장 왼쪽에 있는 노드에서부터 Cumulate_subtrie 배열의 인덱스를 0번으로 할당하며, 각각의 노드에 대한 인덱스에 해당하는 Cumulate_subtire에서의 값은 해당 노드에 존재하고 있는 가장 왼쪽에 위치한 노드가 형성하게 될 서브트라이 이전에 존재할 수 있는 서브트라이의 개수이다. 예를 들어, [그림 5]에서 0번 서브트라이에서 11번 노드는 인덱스 6번을 받게 되며, 이때 11번 노드의 첫 번째 자식노드가 형성하게 될 서브트라이 이전에 존재하는 서브트라이는 0번, 1번, 2번 서브트라이로 3개의 서브트라이가 존재하므로, Cumulate_subtrie의 6번 인덱스에 저장된 값은 3이 된다. Cumulate_subtrie과 External_bitmap의 값을 이용하여, 검색과정에서 검색을 진행할 서브트라이를 찾을 수 있으며, 이는 '3.검색과정'을 통해 자세히 설명하겠다,

- Rule_pointer

내부비트맵이 1인 경우에 해당하는 노드에 저장된 룰 중에 가장 우선순위(priority)가 높은 룰이 저장되어있는 외부메모리의 룰 테이블의 주소값이 Rule_pointer배열에 저장되어있다.

- Start_pointer

접근한 서브트라이의 내부비트맵 배열(Internal_bitmap[ ])에서 처음으로 1의 값을 갖는 노드에 대한 Rule_pointer 주소값을 Start_pointer에 저장한다.

3.2 구축과정(Build)

리프-푸싱 된 영역분할 사분트라이에서의 칩-내부메모리 접근횟수는 칩-내부메모리에 저장되어 있는 트라이의 노드에 접근하는 경우이기 때문에, 노드 접근 횟수가 칩-내부메모리 접근횟수라고 할 수 있다. 하지만, 비트맵트라이를 적용함으로서, 칩-내부메모리에는 노드가 아닌, 배열형태로 되어있는 서브트라이가 저장된다.

하나의 서브트라이에는 크게 내부비트맵과 외부비트맵이 존재한다. 제안하는 구조의 서브트라이에는 2개의 레벨이 존재하므로, stride 개수는 2이고, 2차원 검색을 진행하기 때문에 1개의 노드에는 최대 4개의 자식노드가 존재할 수 있으므로, N_{child}는 4이다. [표 1]에 의해 내부비트맵은 21비트이고, 외부비트맵은 64비트가 된다.

[그림 5]에 리프-푸싱 된 영역사분트라이 기반 비트맵 트라이의 메모리 구조를 나타내었다. 해당 메모리 구조는 크게 칩-내부메모리(on-chip memory)와 칩-외부메모리(off-chip memory)로 나누어 볼 수 있다. 칩-외부메모리에는 룰의 IP주소 정보를 제외한 나머지 4가지 필드에 대한 정보가 저장되어 있으며, 같은 노드에 저장된 룰들에 대해서는 서로 링크드리스트(linked list)로 연결되어있다. 칩-내부메모리에는 크게 서브트라이와 룰 포인터(rule_pointer)가 저장되는데, 서브트라이 안에는 내부비트맵(Internal_bitmap), 외부비트맵(External_bitmap), 누적서브트라이(Cumulate_subtrie), 시작포인터(start_pointer)가 포함된다. 룰 포인터는 내부비트맵의 1에 해당하는 룰이 저장되어 있는 칩-외부메모리의 주소가 저장되어 있다. 서브트라이에서 시작포인터는 해당 서브트라이에서 첫 번째 룰 포인터의 위치를 나타내는 주소 값이 저장되어 있다. 예를 들어, 6번 노드의 경우, 0번 서브트라이에서 내부비트맵 상 두 번째 1에 해당되는 노드이므로, 시작포인터가 가르치는 엔트리의 다음 엔트리에 저장되어 있는 주소값을 따라 칩-외부메모리에 접근하여 룰 검색(rule search)을 진행하게 된다. 칩-외부메모리의 각 엔트리

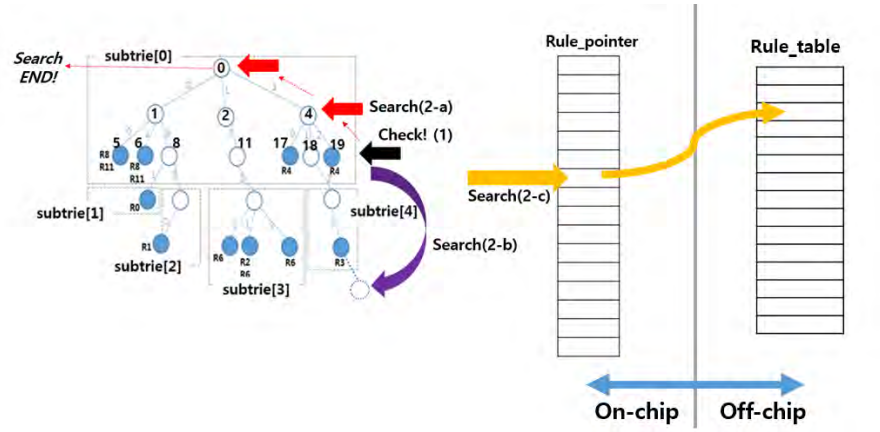

그림 6. 검색과정의 예
Fig. 6. The example of Search procedure

는 3비트로 이루어져 있는데, 처음 1비트는 해당 경로에 자식노드가 존재하는지에 대한 여부를 나타내고, 자식노드가 있을 경우에 대해서 1, 없는 경우에 대해서 0을 할당한다. 나머지 2비트는 해당 자식노드가 부모노드의 몇 번째 노드(0~3)인가를 나타낸다. 누적서브트라이는 서브트라이의 마지막 레벨에서의 해당 노드 이전에 존재하는 서브트라이의 개수가 저장되어 있다. 외부비트맵 엔트리의 첫 번째 비트를 제외한 나머지 2비트와 누적서브트라이는 검색과정에서 다음 서브트라이의 번호를 구하는 데에 사용된다.

3.3 검색과정(Search)

리프-푸싱 된 영역분할 사분트라이 기반 비트맵트라이의 검색과정은 접근한 서브트라이에서의 마지막 레벨에서의 내부비트맵과 외부비트맵에 따라서 달라지며, [그림 6]에 각각의 경우를 나타내었으며, [그림 7]에 검색과정에 대한 의사코드(pseudo code)를 나타내었다. 이 때, [그림 6]은 제안하는 구조의 검색진행을 쉽게 설명하기 위하여 리프-푸싱 된 영역분할 사분트라이의 형태로 나타내었지만, 실제 구조에서는 배열 형태로 이루어져 있다.

경우(2-c)) 내부비트맵 1인 경우,

리프-푸싱(Leaf-pushing)의 특성으로 인해, 룰이 저장된 노드는 리프-노드에만 존재한다. 다시 말해, 검색과정에서 리프-노드에 접근했다는 것은 접근한 노드의 상위레벨에는 룰 노드(rule node)가 존재하지 않으며, 접근한 노드의 하위레벨에도 노드가 존재하지 않음을 의미한다. 그러므로, 이 경우, 접근한 룰 노드에 대응하는 룰 포인터(Rule_pointer)의 위치에 저장된 주소에 해당하는 칩-외부메모리의 룰 테이블(Rule_table)로 가서, 입력 패킷과 룰과의 비교를 수행한 후 검색을 종료한다.

경우(2-a)) 내부비트맵 0,외부비트맵 1 경우,

내부비트맵이 0인 경우는 접근한 인덱스에 해당하는 노드가 없거나, 흰 노드(white node)인 경우를 의미한다. 이 경우는 외부비트맵이 1이므로, 다음에 검색을 진행할 서브트라이가 존재함을 의미한다. 그러므로, 여기서의 내부비트맵 0은 흰 노드를 의미하며, 리프-푸싱의 특성으로 인해, 상위레벨에는 룰 노드가 존재하지 않음이 보장되므로, 다음 서브트라이의 마지막 레벨의 노드에 접근하여, 검색을 계속 진행한다.

경우(2-b)) 내부비트맵 0,외부비트맵 0 경우,

이 경우, 더 이상 검색을 진행할 하위레벨의 서브트라이가 존재하지 않는다는 것만이 보장되므로, 접근한 서브트라이의 상위레벨로의 검색을 진행한다. 검색 과정의 한 예를 들어보면, 위의 그림의 트라이에서 입력의 코드워드가 122인 경우를 가정하여 보면, 서브트라이의 번호는 먼저 코드워드 12를 사용했을 때 노드번호 11이고, 11번 노드의 자식노드에 대한 정보는 외부비트맵의 11-(4^0+4^1)=6번 행에 나타나 있고, 서브트라이의 마지막레벨 노드에서 검색을 진행할 경로의 코드워드가 2이므로, 외부비트맵의 2+1=3번째 열을 본다. 이때 (1 0)이 저장되어 있으므로, 해당경로의 자식노드가 존재함을 의미한다. 또한, 외부비트맵의 6번 행을 보았으므로, 이와 같이 누적서브트라이의 인덱스 6번에 저장된 수 3과 외부비트맵의 뒤의 2비트 0(00)을 더해준 것이 다음 서브트라이의 번호가 되며, 이

```
    N_child = 4; //하나의 노드에 대한 최대 자식노드 수
    N_stride = 2; // 하나의 서브트라이 내의 stride갯수
    lv_subtrie = -1; //서브트라이 단위에서의 레벨
    subtrie_num = 0; //서브트라이 번호
    cons = 1 + N_child; // 서브트라이1개에서 마지막 레벨을 제외한 나머지레벨에 존재하는 최대 노드수

while()
{
    lv_subtrie++;

    for(int lv = stride ; lv < 0 ; i--) // 접근한 서브트라이 내에서의 검색진행
    {
        node_num = 0; // 하나의 서브트라이 내에서의 노드번호 초기화 부분

        for(i = 0; i < lv ; i++) //접근하는 노드번호 계산
        {
            node_num = (node_num * N_child) + (input[(lv_subtrie)*(N_stride+1)+ lv] + 1);
        }

        if( lv == stride)
        {
            last_node_num = node_num; //접근한 서브트라이의 마지막레벨에서 접근하는 노드번호기억
        }

        if(subtrie[subtrie_num].Internal_bitmap[node_num]==1) //내부비트맵=1 경우; [그림6]의 경우(2-c)
        {
            Access to Rule_Table in Off-chip memory for rule search;
            SEARCH END;
        }
        else // 내부비트맵=0경우
        {
            if(subtrie[subtrie_num].External_bitmap[last_node_num][input[(lv_stride+1)*(N_stride+1)]][1]==1)
                //내부비트맵=0, 외부비트맵=1경우; [그림6]의 경우(2-a)
            {
                subtrie_num = subtrie[subtrie_num].Cumulate_subrie[last_node_num - cons]
                            +  subtrie[subtrie_num].External_bitmap[last_node_num][2]

                break to access to next subtrie;
            }
            else // 내부비트맵=0, 외부비트맵=0 경우; [그림6]의 경우(2-b)
            {
                continue to accee upper level in the same subtrie;
            }

        }
    }
}
SEARCH END
```

Fig. 7. pseudo code of search procedure
그림 7. 검색과정에 대한 pseudo code

경우 3이므로 3번 서브트라이로 가서 검색을 계속 진행한다.

Ⅳ. 실험 및 성능 평가

성능평가는 크게 메모리 요구량과 메모리 접근 횟수로 나누어 살펴보았다. 실제로 패킷분류가 일어나는 환경과 유사한 환경으로 실험을 진행하기 위해, 실제 라우터에서 사용되고 있는 룰 셋(rule set)과 유사한 특성을 갖는 클레스 벤치(class bench)에서 제공하는 룰 셋인 ACL(Access Control Lists), FW(Firewalls), IPC(IP Chain)를 사용하였다. ACL의 경우, 완전히 구체화된 근원지 주소와 목적지 주소로 이루어진 프리픽스 쌍으로 구성되어 있으며, FW의 경우에는 완전하게 구체화된 목적지 주소와 와일드카드로 표현된 근원지 주소로 이루어진 프리픽스 쌍이 대부분이다. IPC는 프리픽스 쌍을 이루는 근원지 주소와 목적지 주소가 ACL과 FW와는 달리 비교적 다양한 길이에 분포되어 있다.[13] 본 논문의 성능분석을 위해 ACL, FW, IPC에 대해 크기를 1k, 5k, 10k로 크기를 증가시켜가며, 각 성능의 변화양상을 살펴보았다.

4.1 메모리 요구량

전체적인 칩-내부메모리는 서브트라이에 대한 메모리 요구량과 Rule_pointer의 메모리 요구량으로 이루어져 있으며, [표 3]에서 전체 칩-내부메모리와 Rule_pointer의 메모리 요구량을 비교해보면, 서브트라이의 메모리 요구량이 칩-내부메모리의 대부분을 차지하고 있음을 볼 수 있다. 서브트라이를 이루는 요소 중, 내부비트맵(Internal bitmap)과 외부비트맵(External bitmap)은 룰 개수나 노드 개수에 영향을 받지 않고, 서브트라이 안에 있는 stride의 영향을 받으므로, 룰 셋의 크기나 종류에 상관없이 같다.

또한, 다른 서브트라이의 요소인 Cumulate_subtrie

표 2. 칩-외부메모리의 구조 및 메모리 요구량
Table 2. Off-chip memory structure and required amount of memory

		Field	No. of bits
Off-chip Memory	Each rule entry in rule table	Rule number	$\lceil \log_2 N_R \rceil$
		Source prefix length	6
		Source prefix	32
		Destination prefix length	6
		Destination prefix	32
		Source port wild	1
		Source port start	16
		Source port end	16
		Destination port wild	1
		Destination port start	16
		Destination port end	16
		Protocol wild	1
		Protocol type	3
		Next rule pointer	$\lceil \log_2 N_S \rceil$

*N_R : 구축과정에서 쓰인 룰의 개수 / N_S : 룰 테이블에 존재하는 룰의 개수

표 3. 칩-내부메모리의 구조 및 메모리 요구량
Table 3. On-chip memory structure and required amount of memory

		ACL1k	ACL5k	ACL10k	FW1k	FW5k	FW10k	IPC1k	IPC5k	IPC10k
No. of rules		957	4659	9732	869	3064	4349	988	4463	8108
No. of nodes in leaf-pushed AQT		8466	30001	82299	733	7537	477	5951	4437	2581
No. of stored rules		7356	39360	120944	9202	195777	39469	11134	43054	56500
No. of subtrie		2660	9159	24387	237	2449	149	1990	1441	837
No. of cases 'internal bitmap=1'		5208	21421	54049	551	5654	359	3412	1108	1937
Each sub-trie (bit)	Internal bitmap	21	21	21	21	21	21	21	21	21
	External bitmap	192	192	192	192	192	192	192	192	192
	Cumulate_subtrie	192	224	240	128	192	128	176	176	160
	Start_pointer	13	15	16	10	13	9	12	11	11
bits for each subtrie		418	452	469	351	418	350	401	400	384
Rule_pointer (byte)		10963	40165	114855	965	12722	719	5972	2216	3874
Total required on-chip memory in Leaf-pushed AQT Bitmap (byte)		149948	557648	1544542	11363	140682	7237	105720	74266	44050

와 Start_pointer의 경우, 칩-외부메모리에 저장되어 있는 룰의 개수의 영향을 많이 받는다. 그러나, [표 3]을 고려해 보았을 때, Cumulate_subtrie의 경우에는 최소 128비트에서 최대 240비트, Start_pointer의 경우 최소 9비트에서 최대 16비트로 서브트라이 개수의 분포를 고려해 보았을 때, 룰 셋의 종류와 크기에 상관없이 비교적 적게 차이가 남을 볼 수 있다. 하지만 서브트라이의 개수의 경우에는 FW10k의 경우 149개이며, ACL10k의 경우 24387개로, 서브트라이의 개수의 분포는 룰 셋의 종류나 크기의 영향을 많이 받음을 알 수 있다. 서브트라이의 개수는 비트맵 트라이를 적용하기 전의 리프-푸싱 된 영역분할 사분트라이에 존재하는 노드의 개수의 영향을 많이 받는다. ACL의 경우에는 근원지 IP프리픽스와 목적지 IP프리픽스가 모

표 4. 칩-내부메모리 요소로의 접근횟수
Table 4. The number of accesses to on-chip memory elements

		ACL1k	ACL5k	ACL10k	FW1k	FW5k	FW10k	IPC1k	IPC5k	IPC10k
Leaf -pushed AQT Bitmap	Avg Internal bitmap acc	11.228	10.673	11.026	4.109	5.120	4.529	8.251	10.123	7.763
	Avg External bitmap acc	9.699	9.274	9.566	2.749	3.760	3.271	6.930	8.376	6.219
	Avg subtrie acc	10.629	10.169	10.409	3.178	4.248	3.739	7.399	8.690	6.555

표 5. 칩-외부메모리로의 접근횟수
Table 5. The number of accesses to off-chip memory

	ACL1k	ACL5k	ACL10k	FW1k	FW5k	FW10k	IPC1k	IPC5k	IPC10k
Avg on-chip to off-chip acc	0.9998	0.9999	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Avg rule acc	1.883	3.394	1.715	55.248	109.812	172.844	2.125	4.426	37.583

두 길기 때문에, 대부분의 룰 노드가 하위 레벨에 저장되므로 노드의 개수가 많아진다. 반면, FW의 경우에는 룰 셋의 특성상 대부분의 근원지 IP프리픽스가 와일드카드로 되어있기 때문에 영역분할 사분트라이를 형성하는 데에 있어서, 대부분의 룰 노드가 상위 레벨에 존재하기 때문에, 서브트라이의 개수에 있어 ACL의 경우가 FW의 경우의 최소 3.7배에서 최대 163.7배임을 확인할 수 있었다. IPC의 경우, 근원지 IP프리픽스와 목적지 IP프리픽스가 ACL과 FW의 중간형태로 비교적 길이가 다양하게 분포되어 있기 때문에, 서브트라이의 개수가 ACL의 경우가 IPC의 경우의 최소 1.3배에서 최대 29.1배로 FW에 비해서는 많고, ACL에 비해서는 적음을 확인할 수 있었다.

4.2 메모리 접근 횟수

칩-내부메모리에서의 접근으로는 내부비트맵, 외부비트맵, 누적서브트라이로의 접근이 있으며, 칩-외부메모리에서의 접근으로는 칩-내부메모리로부터 칩-외부메모리로의 접근과 룰 테이블에서의 룰 검색을 위한 접근이 있다. ACL, FW, IPC 모두 공통적으로, 평균 내부비트맵 접근횟수는 평균 서브트라이 접근횟수에 비해 약 1번 많고, 평균 외부비트맵 접근횟수는 평균 서브트라이 접근횟수에 비해 약 1번 적음을 확인할 수 있었다. 서브트라이 접근 횟수는 검색이 진행된 모든 서브트라이에 대해 개수를 세어주는 반면, 외부비트맵 접근 횟수의 경우에는 리프-푸싱의 특성에 의해 접근한 서브트라이의 마지막레벨에서의 내부비트맵이 1인 경우에 대해서는 다음 서브트라이로의 접근이 이루어지지 않기 때문에, 외부비트맵에 접근하지 않는다. 이러한 이유로, 외부비트맵 접근횟수의 경우, 평균적으로 서브트라이 접근횟수보다 약1번 적다. 내부비트맵 접근횟수의 경우, 접근한 서브트라이의 마지막 레벨에서의 내부비트맵이 1이고 외부비트맵이 0일 경우에 대해서는 서브트라이 접근횟수보다 1회에서 2회 많은 내부비트맵 접근이 일어나기 때문에, 평균적으로 내부비트맵 접근횟수가 서브트라이 접근횟수보다 약 1회 많다. 같은 룰 셋에 대해서는 룰 셋의 크기와는 상관없이 평균 내부비트맵과 평균 외부비트맵 접근횟수는 평균 서브트라이 접근횟수에 비해 많은 차이가 나지 않았다. 하지만, 룰 셋의 종류가 달라짐에 따라 평균 서브트라이 접근 횟수가 적게는 약 1.479번, 많게는 약 7.451번 많음을 확인할 수 있었다.

위의 [표 5]에서 'Avg on-chip to off-chip acc'는 칩-내부메모리에서 칩-외부메모리로 접근하는 평균횟수를 의미하는데, 제안하는 구조에서는 리프-푸싱으로 인해, 입력패킷에 대한 검색을 진행하는 경로에서 룰 노드를 만나는 횟수는 1번이하이기 때문에, 평균적으로 칩-내부메모리에서 칩-외부메모리로 접근하는 횟수는 최대 1번이다.

'Avg rule acc'는 칩-내부메모리에서의 검색과정에서 룰 노드를 만나 칩-외부메모리 접근했을 때, 룰 검

표 6. 룰 셋 크기 변화에 따른 각각의 노드 갯수와 칩-내부메모리 변화율
Table 6. The change rate of the number of nodes and required on-chip memory according to changes in rule set sizes

		ACL		FW		IPC	
		1k → 5k	5k → 10k	1k → 5k	5k → 10k	1k → 5k	5k → 10k
num of nodes(1k/5k/10k)		8466/ 30001/ 82299		733/ 7537/ 477		5951/ 4437/ 2581	
change rate of nodes		3.544	2.743	10.282	0.063	0.746	0.582
required on-chip memory in 'LP_AQT' (byte) (1k/5k/10k)		62437/ 262509/ 812703		4307/ 57470/ 1250873		43889/ 34387/ 18713	
required on-chip memory in 'LP_AQT_B'(byte) (1k/5k/10k)		149948/ 557648/ 1544542		11363/ 140682/ 7237		105720/ 74266/ 44050	
change rate of required on-chip memory	LP_AQT	4.204	3.096	13.34	21.77	0.783	0.544
	LP_AQT_B	3.719	2.770	12.381	0.051	0.702	0.593

표 7. 룰 셋에 따른 칩 내부 메모리 접근 횟수 비교
Table 7. The comparison of the on-chip memory access

		ACL1k	ACL5k	ACL10k	FW1k	FW5k	FW10k	IPC1k	IPC5k	IPC10k
Avg on-chip acc	LP_AQT	30.878	29.552	30.172	8.586	11.825	10.408	21.314	24.624	18.433
	LP_AQT_B	10.629	10.169	10.409	3.178	4.248	3.739	7.399	8.690	6.555
(LP_AQT) / (LP_AQT_B)		2.905	2.906	2.899	2.702	2.784	2.784	2.881	2.834	2.812

색을 진행하는 평균 룰의 개수와 같으며, 이하 '평균 룰 접근횟수'로 칭하겠다. 평균 룰 접근 횟수의 경우 룰 셋의 종류에 따라 많은 차이를 보인다. ACL의 경우에는 근원지 IP프리픽스와 목적지 IP프리픽스가 모두 구체화 되어 있어서 길이가 길기 때문에, 하나의 노드에 비교적 적은 룰들이 저장되어 있고, 이로 인해 평균 룰 접근 횟수가 최소 1.715에서 최대 3.394번으로 매우 적다. 영역사분 트라이를 형성하는 과정에서 근원지 IP프리픽스와 목적지 IP프리픽스를 한 비트씩 결합하여 하나의 코드워드(codeword)를 형성하는데, 이는 두 IP프리픽스 중 짧은 길이에 맞추어서 형성되므로, 대부분의 근원지 IP프리픽스가 와일드카드인 FW의 경우에는 짧은 코드워드를 사용하여 영역분할 사분트라이를 형성하게 된다. 다시 말해, FW의 경우 대부분의 근원지 IP프리픽스는 와일드카드이고 목적지 IP프리픽스는 길기 때문에, 고려되어야할 목적지 IP프리픽스가 많은데도 불구하고 충분히 고려되지 못한 상태로 노드에 룰이 저장된다. 그러므로, 하나의 노드 안에 많은 룰이 저장되므로, 룰 검색을 진행하는 데에 있어서 ACL에 비해 최소 16.2배에서 최대 100.8배 정도이다. IPC의 경우에는 IP프리픽스에 정보가 ACL과 FW에 비해서 길이가 고르게 분포되어 있기 때문에, 룰 접근 횟수는 ACL에 비해서는 많고, FW에 비해서는 적다.

Ⅴ. 성능 비교

리프-푸싱된 영역분할 사분트라이를 'LP_AQT', 리프-푸싱된 영역분할 사분트라이 기반 비트맵트라이를 'LP_AQT_B'로 줄여서 쓰도록 하겠다. [표 6]은 Ⅳ장의 결과를 바탕으로, 각 룰 셋의 크기가 1k에서 5k, 5k에서 10k로 변화함에 따라 LP_AQT의 노드 개수의 변화율에 따른 결과를 나타낸 것이다. [표 6]에서, 비트맵트라이가 적용된 LP_AQT_B가 LP_AQT에 비해 내부 메모리 요구량이 많았다. 하지만, LP_AQT의 경우, 칩-내부메모리요구량의 변화율이 LP_AQT_B에 비해 컸으며, 노드 갯수의 변화율과도 차이가 컸다. 반면, LP_AQT_B에서의 칩-내부메모리 변화율은 대부분의 경우에서 LP_AQT의 노드 갯수 변화율과 유사했다. 이를 통해, LP_AQT_B에서의 메모리 변화율이 노드개수의 변화율에 비례하여, 안정적으로 변화함을 알 수 있었다. 데이터 셋의 증가에 따

라 요구되는 메모리의 변화량이 안정적이라는 것은 확장성이 좋다는 것을 의미하므로, IPv6로의 변화를 고려 해 볼 때, 메모리 요구량의 변화율은 성능을 평가함에 있어서, 고려할만한 가치가 있다.

LP_AQT와 LP_AQT_B 모두 리프-푸싱의 특성이 반영되었기 때문에, 평균 외부메모리 접근 횟수는 [표 5]의 결과와 같이, 최소 0.9998에서 최대 1.0000번으로 각각의 입력에 대해 약 1번 이하의 외부메모리 접근이 이루어진다. LP_AQT의 칩-내부메모리 접근횟수는 노드 접근횟수와 같다. 하지만 LP_AQT_B의 경우, 여러 개의 비트에 대한 정보인 서브트라이 단위로 데이터를 불러와 처리하기 때문에, 칩-내부메모리 접근횟수는 서브트라이 접근횟수와 같다. 이러한 이유로, [표 7]를 통해 평균 칩-내부메모리 접근횟수가 LP_AQT_B의 경우, ACL은 평균 10.4번, FW는 3.7번, IPC는 7.5번인 반면, LP_AQT의 경우, ACL은 평균 30.2번, FW는 10.3번, IPC는 21.5번으로, LP_AQT_B가 LP_AQT에 비해 평균 칩-내부메모리 접근횟수가 2.7에서 2.9배 적었다.

Ⅵ. 결 론

인터넷 트래픽의 수와 종류가 다양해짐에 따라, 라우터는 더 빠르게 패킷분류를 수행해야 한다. 본 논문에서는 패킷분류를 위한 포워딩 테이블의 메모리를 가능한 적게 하면서도, 속도는 빠르게 하고자, 리프-푸싱된 영역분할 사분트라이 기반 비트맵 트라이(LP_AQT_B)를 제안하고, 리프-푸싱된 영역분할 사분트라이(LP_AQT)의 성능과 비교하였다. 성능 비교 요소로는 크게 메모리 요구량과 평균 메모리 접근횟수 측면으로 살펴보았다. 그 결과, 구축하고자 하는 룰의 개수가 증가함에 따라 LP_AQT의 경우, 메모리 요구량이 불안정한 비율로 구축하고자 하는 룰의 개수 증가율보다 더 큰 비율로 변화함을 보인 반면, LP_AQT_B의 경우, 구축하고자 하는 룰의 개수 증가율에 비례하여, 안정적으로 변화함을 볼 수 있었다. 평균 칩 외부 메모리 접근 횟수의 경우, LP_AQT와 LP_AQT_B 두 경우 모두 리프 푸싱의 특성으로 인해, 1번 이하이다. 하지만 평균 칩 내부 메모리 접근 횟수의 경우는, 두 경우에서 평균 칩 매부 메모리에 접근하는 경우를 다르게 보고 있기 때문에 차이가 난다. LP_AQT의 경우는 노드에 접근하는 횟수를 의미하며, LP_AQT_B의 경우는 서브트라이에 접근하는 경우를 의미한다. 그 결과, 평균 칩 내부 메모리 접근 횟수는 LP_AQT의 경우가, LP_AQT_B의 경우의 최소 2.702배에서 최대 2.906배로, LP_AQT_B의 경우가 LP_AQT의 경우에 비해 칩 내부 메모리 접근 횟수가 확연히 적음을 확인 할 수 있었다.

References

[1] W. Lee, C.-H. Choi, and S.-M. Kim, “Point-to-multipoint services and hierarchical QoS on PBB-TE system,” *J. KICS,* vol. 37B, no. 06, pp. 433-44, 2012.

[2] K.-S. Shim, S.-H. Yoon, S.-K. Lee, S.-M. Kim, W.-S. Jung, and M.-S. Kim, “Automatic generation of snort content rule for network traffic analysis,” *J. KICS,* vol. 40, no. 04, pp. 666-677, 2015.

[3] Y. Choe, M. Shim, and H. Lim, “Packet classification using an area-based quad-trie conditionally merged with decision trees,” *J. KISS,* vol. 41, no. 1, pp. 33-47, Feb. 2014.

[4] P. Gupta and N. McKeown, “Algorithms for packet classification,” *IEEE Network*, vol. 15, no. 2, pp. 24-32, Mar.-Apr. 2001.

[5] H. J. Chao, “Next generation routers,” in *Proc. IEEE*, vol. 90, no. 9, pp. 1518-1588, Sept. 2002.

[6] S.-H. Oh, S.-G. Na, and J.-S. Ahn, “Bit-map trie for fast routing lookups,” in *Proc. KISS Spring Conf.*, vol. 27, no. 1, pp. 328-330, Mar. 2000.

[7] F. Baboescu, S. Singh, and G. Varghese, “Packet classification for core router: is there an alternative to CAMs?,” in *Proc. IEEE INFOCOM*, vol. 1, pp. 53-63, Mar.-Apr. 2003.

[8] M. M. Buddhikot, S. Suri, and M. Waldvogel, “Space decomposition techniques for fast layer-4 switching,” in *Proc. Conf. Protocols for High Speed Netw.*, vol. 31, pp. 25-41, Aug. 1999.

[9] H. Yu and R. Mahapatra, “A Memory-efficient hashing by multi-predicate bloom filters for packet classification,” in *Proc. IEEE INFOCOM*, pp. 2467-2475, Phoenix, AZ, Apr. 2008.

[10] H. Lim, K. lim, and N. Lee, “On adding bloom filters to longest prefix matching

algorithms," in *Proc. IEEE Trans. Comput.*, vol. 10, no. 99, Aug. 2012.

[11] Y. Choe and H. Lim, "Two-dimensional binary search on length using bloom filter for packet classification," *J. KICS,* vol. 37B, no. 04, pp. 245-257, 2012.

[12] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap : Hardware/software IP lookups with incremental updates," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97-122, Apr. 2004.

[13] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499-511, Jun. 2007.

서 지 희 (Ji-hee Seo)

2014년 2월 : 이화여자대학교 전자 공학과 졸업

2014년 3월~현재 : 이화여자대학교 전자공학과 석사과정

<관심분야> Router나 switch등의 Network관련 SoC설계, TCP/IP 관련 하드웨어 설계

임 혜 숙 (Hye-sook Lim)

1986년 : 서울대학교 제어계측공학과 졸업(학사)

1991년 : 서울대학교 제어계측공학과 졸업(석사)

1996년 : The university of Texas at Austin, Electrical and Computer Engineering 졸업(박사)

1996년 11월~2000년 7월 : Lucent Technologies, Member of Technical Staff

2000년 7월~2002년 2월 : Cisco Systems, Hardware Engineer

2002년 3월~현재 : 이화여자대학교 전자공학과 정교수

2014년 : 미래창조과학부 장관상 올해의 여성과학기술인상 수상

<관심분야> 라우터나 스위치 등의 네트워크장비 설계 관련 알고리즘 및 하드웨어 구조 설계, 콘텐츠 중심 네트워크 (CCN), 소프트웨어 정의 네트워크 (SDN).