IPv6 주소 검색을 위한 블룸 필터를 사용한 레벨에 따른 이진 검색 구조

준회원 박 경 혜, 종신회원 임 혜 숙*

Binary Search on Levels Using Bloom Filter for IPv6 Address Lookup

Kyong Hye Park *Associate Member*, Hyesook Lim* *Lifelong Member*

요 약

IPv6는 32 비트를 갖는 IPv4의 주소 공간 부족 문제를 해결하기 위하여 제안된 새로운 IP주소 체계로서 128비트를 갖는다. 그러므로 IPv6의 라우팅 테이블을 트라이 구조에 저장한다고 가정할 때, IPv4에 비해 매우 많은 레벨이 존재하게 된다. 따라서 IPv6 주소 검색을 위해서는 트라이 레벨에 따른 선형 검색보다 레벨에 따른 이진 검색 구조가 적합하며, 검색 성능이 더 우수하다는 장점이 있다. 본 논문에서는 IPv6를 위한 트라이 레벨에 따른 새로운 이진 검색 알고리즘을 제안한다. 본 논문에서 제안하는 구조는 레벨에 따른 이진 검색의 수행 시 통합 블룸 필터를 사용하여 노드가 존재하지 않는 레벨을 미리 걸러주는 방법을 통하여 외부 메모리 접근 횟수를 줄인다. 실제 라우터에서 사용하는 IPv6 라우팅 데이터를 사용하여 시뮬레이션을 수행하였으며, 1096개의 엔트리를 갖는 라우팅 테이블에 대하여 평균 1~3의 메모리 접근을 통하여 IPv6 주소 검색이 가능함을 보았다.

Key Words : IPv6, 주소 검색, 블룸 필터, 레벨에 따른 이진 검색, 최장 길이 프리픽스 일치

ABSTRACT

IP version 6 (IPv6) is a new IP addressing scheme that has 128-bit address space. IPv6 is proposed to solve the address space problem of IP version 4 (IPv4) which has 32-bit address space. For a given IPv6 routing set, if a forwarding table is built using a trie structure, the trie has a lot more levels than that for IPv4. Hence, for IPv6 address lookup, the binary search on trie levels would be more appropriate and give better search performance than linear search on trie levels. This paper proposes a new IPv6 address lookup algorithm performing binary search on trie levels. The proposed algorithm uses a Bloom filter in pre-filtering levels which do not have matching nodes, and hence it reduces the number of off-chip memory accesses. Simulation has been performed using actual IPv6 routing sets, and the result shows that an IPv6 address lookup can be performed with 1-3 memory accesses in average for a routing data set with 1096 prefixes.

I. 서 론

현재 인터넷의 속도는 기하 급수적으로 증가하고 있으며, 라우터로 입력되는 패킷의 속도에 맞추어 선-속도(wire-speed)로 패킷을 포워딩하는 것이 매우 중요하다. 따라서 많은 알고리즘들이 라우터의 성능을 높이기 위해 개발되고 있으며, 라우터의 성능 평가를 위해 다음과 같은 척도가 사용되고 있다. 첫째,

※ This research was partly supported by the MIC(Ministry of Information and Communications) under a HNRC-ITRC support program supervised by IITA(Institute of Information Technology Assessment).

* 이화여자대학교 SoC Desing Lab. (hlim@ewha.ac.kr)

논문번호 : KICS2008-02-071, 접수일자 : 2008년 2월 11일, 최종논문접수일자 : 2009년 3월 16일

라우터에 저장된 라우팅 테이블에 대한 주소 검색 시에 소요되는 메모리 접근 횟수이다. 적은 메모리 접근 횟수로 포워딩 정보를 찾아야 빠른 패킷 포워딩이 가능하다. 둘째, 라우터에 저장되는 라우팅 테이블의 크기이다. 한정된 라우터의 메모리에 가능한 많은 프리픽스를 효율적으로 저장할 수 있어야 하며, 이는 IP 주소 검색 알고리즘의 데이터 구조에 의존한다. 셋째, 라우터에 연결된 네트워크들의 프리픽스 정보를 쉽게 추가 또는 삭제할 수 있는 테이블 갱신의 용이성이다. 실제로 초당 많은 수의 프리픽스가 갱신되고 삭제된다. 따라서 이와 같은 정보를 실시간으로 수용하여 검색이 정확히 이루어 질 수 있도록 하는 것이 중요하다. 넷째, IPv4 주소 체계에서 IPv6 주소 체계로의 확장성이다. 현재 IPv4의 주소가 보편적으로 사용되고 있으나 포화 상태에 이르고 있어, 무한대에 가까운 인터넷 주소를 사용할 수 있는 IPv6 주소 체계로의 이동이 불가피하다. 따라서 이와 같은 변화를 수용할 수 있어야 한다.

과거에는 클래스를 갖는 주소 체계에 의해 확정 일치(exact match) 방법으로 검색을 진행하였으나, 프리픽스의 낭비를 막기 위해 현재에는 CIDR(classless inter-domain routing) 방식이 사용되고 있으며, 이로 인해 네트워크에 연결된 호스트의 수에 따라 다양한 크기의 네트워크를 구성할 수 있게 되었다. 따라서 다양한 길이의 프리픽스가 생겨나게 되었으며, 라우터와 같은 스위칭 장비에서는 인터넷 주소 검색에 있어 검색 방법이 매우 복잡해 졌다. 즉, 입력 패킷의 목적지 주소의 네트워크 부분인 프리픽스 길이에 대한 정보를 미리 알고 있지 않으므로 하나의 입력 패킷에 대해 라우터에 존재하는 다수의 프리픽스가 일치할 수 있고, 이 중 가장 길게 일치하는 프리픽스(longest matching prefix, LMP)가 최적으로 일치하는 프리픽스(best matcing prefix, BMP)가 된다[1].

본 논문에서는 길이가 긴 IPv6 주소 검색에 적합하다고 판단되는 레벨에 따른 이진 검색 구조를 적용하여 노드 정보를 생성하고 이 정보를 통합 블룸 필터와 통합 복수-해쉬 테이블에 저장하여 효율적으로 IP 주소 검색을 수행하는 구조를 제안한다.

본 논문의 구성은 다음과 같다. 2장에서는 기존의 IP 주소 검색 알고리즘에 대하여 간단히 소개한다. 3장에서는 본 논문에서 제안하는 IPv6 주소 검색을 위한 블룸 필터를 사용한 레벨에 따른 이진 검색 구조에 대하여 살펴보며, 4장에서는 기존의 알고리즘과의 성능비교를 한다. 마지막으로 5장에서 간단히 결론을 맺는다.

Ⅱ. 기존의 구조

이절에서는 본 논문에서 제안하는 구조와 관련이 있는 트리 구조에 기반한 알고리즘, 해쉬에 기반한 알고리즘, 블룸 필터를 기반으로 하는 알고리즘, 마지막으로 기타 알고리즘에 대하여 간단히 설명한다. 표 1에 프리픽스 집합의 예를 보였다. 각 알고리즘을 표 1의 프리픽스 집합을 예로 하여 설명한다.

2.1 트리 구조에 기반한 알고리즘

2.1.1 이진 트라이(Binary Trie, B-Trie)[2]

가장 기본적인 구조로써 프리픽스의 각 비트 값을 경로로 하여 해당하는 노드에 정보를 저장하는 방식이다. 즉, 프리픽스의 최상위 비트(most significant bit, MSB)부터 진행하며, 트리의 루트부터 각 비트 값을 확인한다. 만일 비트 값이 0이면 현재 노드에서 왼쪽으로, 1이면 오른쪽으로 이동하며 프리픽스의 모든 비트를 표현할 수 있는 노드에 다다르면 그 노드에 프리픽스의 포워딩 정보를 저장한다. 이와 같은 방법으로 표 1의 프리픽스를 이용하여 이진 트라이를 구성하면 그림 1과 같다.

이진 트라이에서의 검색 과정은 다음과 같다. 입력 패킷의 최상위 비트부터 트라이의 루트 노드에 서 검색을 진행한다. 저장 과정과 마찬가지로 해당 비트가 0이면 트라이의 왼쪽으로, 1이면 오른쪽으로 검색을 진행한다. 만일 프리픽스 노드를 만난다면 그 노드에 저장된 포워딩 정보를 기억하며 검색을 계속 진행한다. 만일 이진 트라이의 잎(leaf) 노드에 도달하면 검색이 종료되며, 그 순간까지 기억하고 있던 포워딩 정보가 최장 길이 일치 프리픽스(longest matching prefix, LMP)의 포워딩 정보이므로 이 값을 최종적으로 출력하게 된다.

하지만 이진 트라이의 경우 구성에 있어서는 간단하지만 빈 노드가 많이 존재하게 되어 다른 알고리즘에 비해 메모리 요구량이 많다는 단점이 있다. 또

표 1. 프리픽스 집합

	Prefix
P0	0011*
P1	01*
P2	1*
P3	100*
P4	11*
P5	1110*
P6	11100*

한 검색 속도에 있어서도 최악의 경우 프리픽스의 최대 길이까지 선형적으로 검색을 진행하여야 하므로, 검색 성능도 우수하지 못하다는 단점이 있다. 특별히 이진 트라이는 최대 128비트를 갖는 IPv6 의 주소 검색을 위해서는 메모리 요구량이나 검색 성능의 면에서 적절치 못한 것으로 판단된다.

2.1.2 레벨에 따른 이진 검색(Binary Search on Levels, BSL)[3]

레벨에 따른 이진 검색은 Waldvogel이 제안한 구조이다[3]. 레벨에 따른 이진 검색은 프리픽스 집합을 저장한 트라이의 레벨을 이용하여 이진 검색을 수행하되, 일치하는 노드가 있으면 더 긴 레벨로, 일치하는 노드가 없다면 더 짧은 레벨로 검색을 진행하는 이진 검색 구조이다. 각 레벨에 존재하는 노드들은 레벨별로 별도의 해쉬 테이블에 저장되어야 하는데, 프리픽스가 존재하는 레벨 만을 뽑고 각 레벨별로 완전해싱 함수가 주어짐을 가정하여, 해당 레벨의 모든 노드에 대한 해쉬 인덱스를 구하여 해쉬 인덱스가 가리키는 엔트리에 노드를 저장한다.

표 1의 프리픽스 집합으로 레벨에 따른 이진 검색 구조를 구성하면 그림 2와 같다. 그림 2의 오른쪽에는 레벨에 따른 접근 순서를 보였다. 프리픽스가 존재하는 트라이 레벨 중 가운데 존재하는 레벨인 레벨 3가 가장 먼저 접근되며, 레벨 2나 레벨 4가 그 다음 접근 레벨이며, 레벨 1 혹은 레벨 5가 가장 나중에 접근되는 레벨이다. 그림 2와 그림 1을 비교하여 보면, 그림 2는 이진 트라이의 빈 노드에 마커나 그 노드까지의 BMP를 미리 계산하여 저장하였다는 점이 다르다. 마커나 BMP는 이진 검색 시 마지막으로 접근하는 레벨(그림 2에서는 레벨 1과 레벨 5)을 제외한 나머지 레벨에 존재하는 빈 노드에 저장

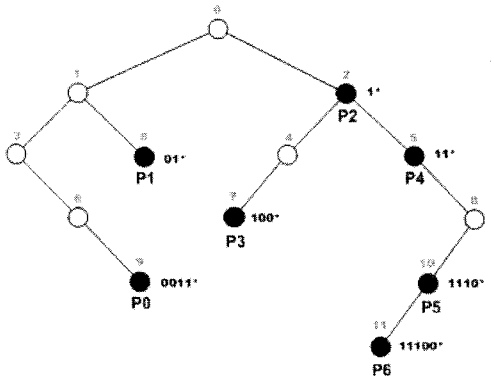


그림 1. 이진 트라이(Binary Trie, B-Trie)

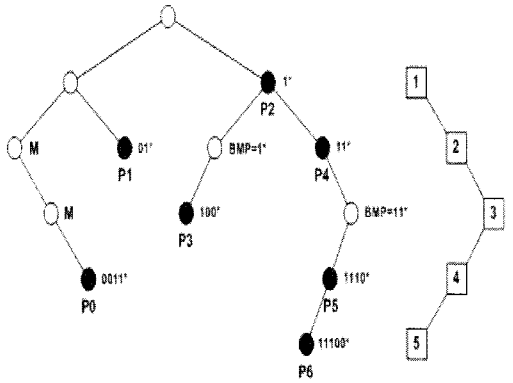


그림 2. Waldvogel의 레벨에 따른 이진 검색(binary search on levels)

된다. 그 이유는 트리의 루트로부터 경로를 따라 검색이 진행되는 것이 아니라 프리픽스 길이를 이용하여 레벨 별로 검색이 진행되기 때문에 현재 접근한 레벨의 윗 레벨과 아래 레벨에 어떤 정보가 저장되어 있는지 알 수 없기 때문이다. 즉, 빈 노드에 접근하였다고 가정하였을 때, 일치하는 값이 없다고 생각하여 짧은 레벨 쪽으로 검색을 진행하지만 실제로는 입력된 주소와 일치하는 길이가 긴 프리픽스가 아래 레벨에 존재 할 수 있기 때문이다.

따라서 현재 접근한 노드의 레벨에는 프리픽스가 존재하지 않으나 현재 레벨보다 길이가 긴 쪽에 프리픽스 정보가 있고 검색을 아래 레벨로 진행해야 한다는 뜻으로 마커를 삽입한다. 하지만 만일 현재 접근 길이에 프리픽스가 존재하지 않으나 마커가 존재하여 검색을 아래 레벨로 진행하였을 때, 일치하는 프리픽스가 존재하지 않을 수도 있다. 이 경우 다시 마커가 존재하던 이전 접근 레벨보다 짧은 쪽 레벨로 계속 검색을 진행하여야 하는 백-트래킹(back-tracking)이 발생하게 된다. 따라서 이것을 방지하기 위해 빈 노드에 이 노드에 해당할 때의 BMP 정보를 미리 계산하여 저장한다. 검색이 진행되는 동안 현재까지의 BMP 정보를 계속 기억하며 더 이상 검색을 진행할 레벨이 존재하지 않는 경우 현재까지 기억하고 있는 BMP에 대한 포워딩 정보를 출력하고 검색을 종료할 수 있도록 한다. 여기서 각 레벨을 접근하는 방법은 각 레벨별로 완전 해싱 함수가 주어짐을 가정하여 어떤 인풋 목적지 주소가 주어졌을 때, 주어진 인풋을 해당 레벨의 길이만큼만 잘라 해쉬 인덱스를 구하여, 그 인덱스가 가르키는 엔트리에 저장된 값을 이용한다.

레벨에 따른 이진 검색은 최대 검색 횟수가 다른

알고리즘에 비해 매우 적어, 길이가 긴 IPv6 주소 검색에 적합한 것으로 판단된다.

2.1.3 이진 검색 트리(Binary Search Tree, BST)[4]

이진 검색 트리는 이진 트라이와 달리 빈 노드가 없어 메모리 요구량이 적으며, 트리의 깊이가 이진 트라이에 비해 감소한다는 점이 특징이다. 이 알고리즘에서는 프리픽스의 크기를 비교하는 정의를 세워, 이를 기준으로 하여 프리픽스를 크기별로 정렬한 후, 정렬 된 값 중 가장 중간 값을 각 트리의 노드로 지정하여 트리를 구성하게 된다. 표 1의 프리픽스 집합을 이용하여 이진 검색 트리를 구성한 결과는 그림 3과 같다.

프리픽스의 크기는 다음과 같이 정의된다. 두 프리픽스의 길이가 같은 경우에는 수학적 값을 비교한다. 만일 길이가 다른 경우에는 짧은 프리픽스를 기준으로 수학적 값을 비교한다. 이 때 수학적 비교 값이 같은 경우에는 길이가 긴 프리픽스의 다음 비트를 확인하며 그 값이 0이면 길이가 짧은 프리픽스가 큰 프리픽스로, 1이면 긴 프리픽스가 큰 프리픽스로 정의된다.

하지만 이런 크기에 대한 정의만을 사용하여 프리픽스들을 정렬한 후, 이진 검색을 수행한다면, 프리픽스 사이의 네스팅(nesting) 관계가 고려되지 않아 검색이 잘못 진행 될 수 있다. 이진 검색 트리를 구성하기 위하여 각 프리픽스를 인클로져(enclosure), 인클로즈드(enclosed), 디스조인트(disjoint)로 분류하며, 인클로져는 자신을 부-스트링(sub-string)으로 갖는 프리픽스가 존재하는 프리픽스이며, 인클로즈드는 인클로져를 부-스트링으로 갖는 프리픽스, 디스조인트는 다른 프리픽스와 네스팅 관계를 갖지 않는 프리픽스이다. 이와 같은 정의를 사용하여 이진 트리 검색 시 인클로져 프리픽스가 인클로즈드 프리픽스보다 먼저 검색되게 하여 검색이 올바르게 진행되도록 한다. 따라서 이진 검색 트리는 프리픽스의 네스팅 관계를 고려하여 구성하므로, 프리픽스의 네스팅 정도에 따라 불균형이 심해질 수 있다는 단점이 있다.

2.1.4 영역 분할 이진 검색(Binary Search on Range, BSR)[5]

영역 분할 이진 검색은 프리픽스를 영역으로 표현하여, 그 영역에 해당하는 프리픽스의 포워딩 정보를 저장한 구조이다. IPv4 주소의 경우, [0, 2^{32}-1] 영역에서의 한 점으로 표현되며, 각 프리픽스를 최대 길이인 32비트로 확장하여 프리픽스에 기반한 영역을

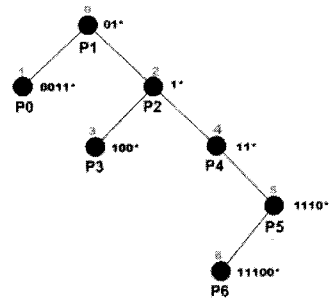


그림 3. 이진 검색 트리(Binary Search Tree, BST)

생성한다. 하나의 프리픽스는 두 개의 점으로 영역을 표현하며, 0으로 채운(padding) 값은 영역의 시작점(start point)으로, 1로 채운 값은 영역의 끝점(end point)으로 정의한다. 표 1의 프리픽스의 예에 대하여 최대 길이를 5 비트로 가정하였을 때, 위와 같은 방법을 사용하여 영역으로 표현한 결과는 그림 4와 같다.

영역 분할 이진 검색 알고리즘은 공통 부분을 갖지 않는(disjoint) 영역의 각 점을 기준으로 하여 라우팅 테이블의 엔트리를 구성한다. 각 엔트리보다 큰 경우와 같은 경우에 대하여 BMP를 선-계산(pre-compute)하여 라우팅 테이블에 저장한다. 그림 4에 표현된 영역 분할 이진 검색 알고리즘을 사용하여 라우팅 테이블을 구성하면 그림 5와 같다.

그림 5의 라우팅 테이블에서 이진 검색을 이용하여 입력 패킷과 가장 길게 일치하는 프리픽스를 검색한다.

2.2 해쉬 구조에 기반한 알고리즘

2.2.1 병렬 해쉬 구조(Parallel Hashing Architecture)[6]

병렬 해쉬 구조에서는 프리픽스 길이 별로 각각 해쉬 하드웨어, 주-테이블, 보조 테이블을 갖는다. 모든 테이블은 각각 별도의 SRAM에 저장되어 있으므로 검색 시 각 테이블에 병렬적으로 접근하여 검색을 하며, 우선 순위 인코더에서 각 테이블에서의 결과 중 가장 길이가 긴 결과를 최종 검색 결과로 출력하게 된다. 이와 같은 방법으로 확정 일치(exact matching)를 사용하여 검색을 진행할 수 있다.

각 프리픽스 길이의 수에 따라 주-테이블과 보조

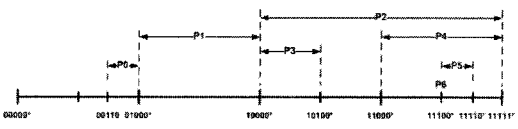


그림 4. 영역으로 표현된 프리픽스

값(value)	BMP (엔트리 값보다 클 때)	BMP (엔트리 값과 같을 때)
00110	P0	P0
00111	-	P0
01000	P1	P1
01111	-	P1
10000	P3	P3
10011	P2	P3
11000	P4	P4
11100	P5	P6
11101	P4	P5
11111	-	P4

그림 5. 영역 분할 이진 검색 알고리즘의 라우팅 테이블

테이블, 해싱 함수를 구성하며, 각 프리픽스에 해당하는 해싱 색인을 구하여 그 값을 포인터로 하여 주-테이블에 저장한다. 만일 그 위치에 다른 값이 저장되어 있어 충돌(collision)이 발생한다면, 충돌이 발생한 주-테이블의 엔트리가 가리키는 보조 테이블에 정보를 저장하고, 주-테이블에 충돌이 발생한 횟수를 기록하여 보조 테이블에 정보가 저장되어 있다는 표시를 한다. 그림 6에서 병렬 해싱의 전체 구조를, 그림 7에서는 주-테이블과 보조 테이블의 엔트리 구조를 나타내었다. IPv4 주소의 경우 프리픽스의 길이가 8~32의 분포를 가지므로 해싱 함수, 주-테이블, 보조 테이블은 각각 25개를 갖는다.

병렬 해싱 구조에서의 검색 과정을 살펴보면 다음과 같다. 입력 패킷에 대하여 길이 별 해싱 함수를 적용하여 해싱 색인을 구한다. 이 색인 값을 주-테이블을 가리키는 포인터로 이용하여 각 길이 별 주-테이블에 병렬적으로 접근한다. 입력 패킷 값과 일치하는 정보가 주-테이블에 저장되어 있다면 그 값을 저장한다. 만일 일치하는 정보가 존재하지 않으나 현재 접근한 엔트리에서 충돌이 발생하여 보조 테이블에 저장된 정보가 있다면, 현재 엔트리가 가르키는 보조 테이블에서 이진 검색을 수행한다. 최종적으로 각 길이 별 테이블에서 일치한 정보 중 가장 길이가 긴 결과 값을 우선 순위 인코더(priority encoder)에서 선택하여 최종 검색 결과로 결정하게 된다.

이 구조는 병렬적으로 빠르게 검색을 진행하기 위하여 하드웨어 소모가 많고 여러 개의 메모리를 요구한다는 특징이 있으며, 각 라우팅 집합에 맞추어 하드웨어를 구성하므로 소프트웨어적으로 유연하게 설계하기 어려운 단점이 있다.

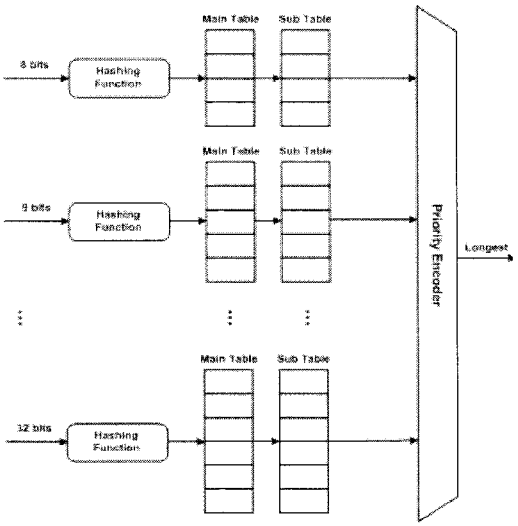


그림 6. 병렬 해싱 구조

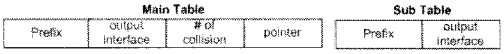


그림 7. 주-테이블과 보조 테이블의 엔트리 구조

2.2.2 복수 해싱(Multiple Hashing)[7]

해싱은 성능을 향상시킬 수 있다는 장점이 있지만, 충돌이 적게 발생하는 해싱 함수를 찾아야 한다. 이와 같이 해싱을 할 때 충돌이 발생하지 않는 완전 해싱(perfect hashing) 함수를 찾기 위해서는 많은 시간이 필요하다는 단점이 있다. 따라서 여러 개의 해싱 함수를 사용하여, 완전 해싱 함수에 근접하는 성능을 보이며, 짧은 시간에 해싱 함수를 찾을 수 있는 방법이 제안되었다. 이 구조를 그림 8에 표현하였다.

그림 8에서 볼 수 있듯이 하나의 프리픽스에 두 개의 해싱 함수를 사용하여 색인을 구하며, 각 색인을 포인터로 사용하여 해당하는 테이블의 엔트리로 접근한다. 두 테이블 중 부하(load)가 적은 쪽에 프리픽스의 포워딩 정보를 저장하는 구조이다. 이와 같은 방법으로 인하여 하나의 해싱 함수를 사용할 때보다 충돌 발생 확률이 현저히 감소하며, 포워딩 정보가 테이블에 균등하게 저장된다는 장점이 있다.

[7]에서는 프리픽스를 16, 24, 32의 길이로 확장한 후, 복수 해싱을 사용하여 해싱 테이블에 저장하

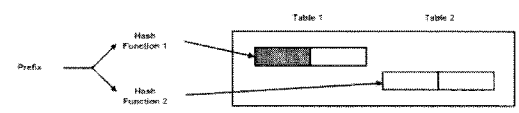


그림 8. 복수 해싱 구조

며, 프리픽스 길이를 레벨로 이용하여 이진 검색을 하는 방법을 제안하였다. 하지만 이 방법은 특정 길이로 프리픽스를 확장하여야 하므로 프리픽스 복사가 불가피하게 일어난다는 단점이 있다. 또한 충돌이 발생하였을 경우를 처리하는 방법에 대한 언급이 없으며, 이 경우 프리픽스 정보를 테이블에 저장하기 위하여 해쉬 함수를 처음부터 다시 찾아야 할 수 있다는 단점이 있다.

2.2.3 병렬 복수 해쉬(Parallel Multiple Hashing)[8]

앞서 언급한 병렬 해쉬 구조에 복수의 해쉬 함수와 해쉬 테이블을 적용한 구조이다. 해쉬 함수를 위하여 CRC 하드웨어를 사용하였다. 병렬 복수 해쉬 구조는 프리픽스를 CRC 하드웨어에 입력하여 해쉬 색인을 구하여 복수 해쉬의 방법으로 테이블에 저장한다. 만일 저장하려는 테이블의 엔트리에 이미 정보가 저장되어 있어 충돌이 일어나는 경우는 작은 크기의 TCAM으로 구성한 오버플로우 테이블에 저장한다.

검색 과정은 다음과 같다. 입력 패킷을 CRC 하드웨어에 입력하여 각 테이블에 해당하는 해쉬 색인을 구한다. 이 값을 각 테이블의 포인터로 하여 병렬적으로 테이블에 접근하며, 찾고자 하는 정보가 저장이 되어 있는지 검색한다. 이때 오버플로우 테이블도 동시에 접근하는 것을 가정하였다. 최종적으로 우선 순위 인코더에서 각 테이블의 결과 값 중 가장 길이가 긴 값을 최종 결과 값으로 결정하게 된다. 병렬 해쉬과 마찬가지로 여러 개의 메모리를 요구하며 하드웨어의 복잡도가 크다는 단점이 있다.

2.2.4 통합 복수 해쉬 테이블[9]

기존의 해쉬 테이블은 길이 별로 구성되어 메모리 요구량이 많고, 구성이 복잡하다는 단점이 있었다. 때문에 특정 길이의 프리픽스를 기준으로 다른 길이의 프리픽스를 확장하여 해쉬 테이블을 구성하는 방식인 통제된 프리픽스 확장(controlled prefix expansion, CPE)[10]의 사용이 제안되었다. CPE를 사용하는 경우 프리픽스의 복사가 불가피하여 저장하여야 하는 프리픽스의 개수가 비효율적으로 많아지는 단점이 있다. 또한 프리픽스의 복사를 피하기 위하여 앞의 몇 비트가 같은 프리픽스끼리 묶어 길이의 종류를 줄이고 그 부분을 따로 처리하는 프리픽스 축약(prefix collapsing)을 이용하여 해쉬 테이블의 수를 줄이는 방법이 제안되었다[11]. 하지만 이러한 방식은 해쉬의 충돌을 증가시켜 비효율적이다.

이 같은 단점을 해결하기 위하여 하나의 단일 테이블에 다양한 길이의 정보를 저장하는 방법이 제안되었다[9]. 통합 복수 해쉬 테이블의 구조는 해쉬 테이블의 개수(해쉬 함수 혹은 해쉬 색인의 개수), 해쉬 테이블당 버켓의 개수 및 버켓당 로드의 개수에 따라 요구되는 메모리의 크기가 달라지고, 또한 오버플로우의 개수가 달라진다. 이 논문에서는 프리픽스의 개수가 N일 때, 두 개의 해쉬 함수를 사용하고, 하나의 테이블당 버켓의 수를 $2^{\lceil \log_2 N \rceil}=N'$, 하나의 버켓당 로드의 수를 2로 갖는 경우에 대한 통합 복수 해쉬 테이블을 보였다. 따라서 해쉬 인덱스에 필요한 총 비트 수는 $\lceil \log_2 N \rceil$이 되며, 각 버켓의 두 로드를 각각 Left와 Right로 정의하였다. 이 두 개의 해쉬 테이블은 각각 SRAM에 저장 가능하다.

프리픽스 정보를 저장하는 과정은 다음과 같다. CRC 하드웨어를 사용하여 $\lceil \log_2 N \rceil$ 비트의 복수의 해쉬 색인을 구하였으며, 각 색인을 포인터로 사용하여 해당하는 테이블의 버켓에 정보를 저장한다. 저장 시 버켓 중 부하가 적은 쪽에 저장하는 것을 원칙으로 하여, 균등하게 저장될 수 있도록 하였다. 만일 두 버킷에 정보가 모두 저장되어 있다면 작은 TCAM으로 이루어진 오버플로우 테이블에 정보를 저장한다.

검색 시에는 기존의 구조와 달리 길이 별 구분이 존재하지 않으므로, 병렬 검색이 어렵다. 따라서 저장되어 있는 정보 중 길이가 가장 긴 것부터 짧은 쪽으로 순차적으로 진행한다. 여기서 오버플로우 프리픽스를 저장하고 있는 오버플로우 테이블은 우선 검색되어 일치하는 프리픽스 정보가 있는지 확인하며, 일치하는 정보가 있다면 그 정보를 미리 인지하고 있음을 가정한다. 먼저 입력 값을 CRC 하드웨어에 넣어 두 개의 해쉬 색인을 얻는다. 해쉬 색인 1의 값으로 Table1의 버킷을 검색하고, 해쉬 색인 2로 Table2의 버킷을 검색한다. 이 때 두 해쉬 테이블은 각각 SRAM에 저장되어 있으므로 병렬로 검색이 진행된다. 검색 중 일치하는 정보가 발견된다면, 미리 찾아진 오버플로우 테이블의 검색 결과와 길이를 비교하여 더 길게 일치하는 결과 값을 선택한다. 이 값이 최장으로 일치하는 프리픽스이므로 그 즉시 검색을 종료하고 최종 결과 값으로 출력한다. 만일 일치하는 프리픽스가 존재하지 않는다면 입력 값을 현재 입력된 길이보다 한 단계 짧은 프리픽스 길이만큼 잘라 검색 과정을 반복한다.

2.3 블룸 필터에 기반한 알고리즘

2.3.1 블룸 필터 이론[10]

블룸 필터는 비트-벡터(bit-vector)를 이용하여 입력 값이 특정 집합의 요소(member)인지 확인하여 필터링을 하는 간단한 구조이다. 블룸 필터를 구성하기 위해서는 프로그래밍이라는 과정을 거쳐야 한다. 블룸 필터의 크기를 M, 블룸 필터에 저장될 요소의 수를 N 이라 정의하였으며, 사용할 해쉬 함수의 개수를 k 로 정의하였다. 총 M-비트로 구성된 블룸 필터에 프로그래밍 하기 위하여 각 해쉬 색인의 값은 1부터 M 사이의 값으로 정의된다. 또한 프로그래밍 되기 전의 블룸 필터의 모든 비트 값은 모두 0으로 초기화 되어있다고 가정한다. 프로그래밍 과정을 보면 다음과 같은데, 즉 어떤 입력에 대하여 k개의 해쉬 함수를 통하여 얻어진 해쉬 색인에 해당하는 블룸 필터의 비트 값을 1로 만든다.

BFAdd(k)
1) for (i = 1 to k)
2) Vector[hi(k)] ← 1

프로그래밍 후 입력 값이 블룸 필터의 요소인지 확인하는 과정을 쿼링(querying)이라 한다. 프로그래밍과 마찬가지로 입력 값에 대하여 k 개의 해쉬 함수에 해당하는 k 개의 해쉬 색인을 구한다. 각 해쉬 색인 값을 포인터로 하여 해당하는 블룸 필터의 비트-벡터에 저장된 값을 확인하며, k 개의 비트-벡터 값이 모두 1이라면 입력 값이 블룸 필터의 요소일 가능성이 있다(positive)는 의미이다. 그러나 비트-벡터 값 중 하나라도 0이 존재한다면 이 입력 값은 블룸 필터의 요소가 아님(negative)을 의미한다. 아래에 블룸 필터에서의 쿼링 과정을 보였다.

BFQuery(k)
1) for (i = 1 to k)
2) if (Vector[hi(k)] = 0) return negative
3) return positive

블룸 필터 이론에서 가장 중요한 점은 쿼링의 결과가 양성(positive)이어도 실제로 그 입력 값이 블룸 필터에 속하는 요소가 아닐 수 있다는 것이다. 즉, k 개의 비트-벡터 값이 모두 1이어도 그 값들은 모두 같은 요소에 의해 프로그래밍 된 것이 아닌 복수의 요소에 의해 프로그래밍 되었을 가능성이 있기 때문이다. 따라서 쿼링의 결과가 양성이며 실제로 그 값이 블룸 필터의 요소라면 참된 양성(true positive), 실제로 요소가 아니라면 거짓 양성(false positive)이라 정의한다. 따라서 좋은 성능의 블룸 필터를 구성하기 위해서는 블룸 필터를 이용하여 필터링을 하였을 때, 잘못 필터링이 되는 거짓 양성의 비율을 줄이는 것이 중요하다.

2.3.2 블룸 필터 기반의 IP 주소 검색 구조[10]

앞서 언급한 블룸 필터를 사용하여 IP 주소 검색을 하는 알고리즘이 제안되었다. 프리픽스 집합에 존재하는 프리픽스 길이 별로 블룸 필터를 구성하는 구조이며, 검색 과정은 다음과 같다. 하나의 입력 값에 대하여 각 블룸 필터에서 쿼링을 진행한다. 우선순위 인코더로부터, 블룸 필터의 결과가 양성인 길이 중 가장 길이가 긴 것부터 선택하여 차례로 해쉬 테이블에서 검색을 진행한다. 만일 해쉬 테이블에 일치하는 결과 값이 있다면, 그 값이 BMP이므로 검색이 즉시 종료된다. 이 같은 방법은 해쉬 테이블에 접근하는 횟수를 감소시켜 검색 성능을 향상시킬 수 있다는 장점이 있으나, 블룸 필터를 길이 별로 구성하여야 하므로 복잡하다는 단점이 있다. 또한 이 구조에서는 해쉬 테이블도 길이 별로 구성하는 것을 가정하였고, 이와 같은 단점을 해결하기 위하여 통제된 프리픽스 확장(controlled prefix expansion, CPE)을 사용하는 방법을 제안하였다[10]. 하지만 이러한 방법은 프리픽스의 복사가 불가피하다. 특히 이 구조는 완전 해쉬 함수를 이용하여 구현하는 것을 가정하였으므로, 실제로 충돌이 발생하였을 때의 해결책을 제시하고 있지 않다는 문제점이 있다.

2.3.3 통합 블룸 필터[12]

기존의 길이 별로 구성되었던 블룸 필터는 구현이 복잡하며 메모리 요구량이 크다는 단점이 있다. 이와 같은 문제점을 개선하기 위하여 단일 블룸 필터에 다양한 길이의 프리픽스를 모두 프로그래밍 하는 통합 블룸 필터가 제안되었다. 이 논문에서는 N개의 프리픽스를 저장하기 위한 블룸 필터 사이즈(M)는 $2^{\lceil \log_2 N \rceil} = N'$으로 정의하였다. 길이가 다양한 입력 값에 대하여 단일 해쉬 하드웨어를 사용하여 여러 개의 해쉬색인을 구하여야 하므로, CRC 하드웨어를 사용한다. 이 논문에서는 해쉬 함수의 개수 k 를 2로 정의하였으며, CRC 하드웨어로부터 각 프리픽스에 해당하는 해쉬 코드를 생성하여 블룸 필터를 프로그래밍하기 위한 길이가 $\lceil \log_2 N \rceil$ 비트인 k 개의 색인을 추출하였다. 해쉬 색인을 이용하여 블룸 필터

를 프로그래밍 하는 과정은 다음과 같다. 블룸 필터의 모든 비트-벡터 값을 0으로 초기화 한 뒤, 각 해쉉 색인이 가리키는 블룸 필터의 비트-벡터 값을 1로 셋팅한다.

블룸 필터에 찾고자 하는 값이 저장되어 있는지 검색하는 쿼링 과정은 다음과 같다. 통합 블룸 필터에서의 쿼링은 기존의 블룸 필터와 같으나, 하나의 블룸 필터에 다양한 길이가 프로그래밍 되었으므로 블룸 필터의 요소 중 가장 긴 길이부터 쿼링을 진행한다. 즉, 블룸 필터에 저장된 정보 중 가장 긴 길이를 이용하여, 입력 값을 자른다. 이 값을 블룸 필터를 프로그래밍 할때와 동일한 CRC 하드웨어에 넣어 2개의 해쉉 색인을 구한다. 두 해쉉 색인 값을 블룸 필터의 비트-벡터를 가리키는 주소로 하여 동시에 접근하며, 두 비트-벡터의 값이 모두 1이라면, 입력 값과 동일한 정보가 블룸 필터에 저장되어 있을 가능성이 있다는 의미를 가진다. 만일 두 비트-벡터 값 중 하나라도 0이 존재한다면, 그 입력 값은 블룸 필터에 저장되어 있지 않다는 것을 의미한다. 이 경우에는 블룸 필터에 저장된 길이 정보 중 현재 접근 길이보다 다음으로 짧은 길이를 이용하여 쿼링을 다시 진행한다.

2.4 기타 알고리즘

현재 인터넷 주소 체계는 IPv4에서 IPv6로 변하고 있다. 가장 주목할 점은 인터넷 주소 길이의 변화인데, 기존의 32비트 주소가 128비트로 급격히 증가하므로, 이 주소 체계에 적합한 알고리즘을 제안하는 것이 중요하다.

[13]에서는 영역 분할 이진 검색 구조를 이용하여, 사용자가 임의로 지정한 비트를 기준으로 시작점(higher-end) 노드, 끝점(lower-end) 노드, 병합(merged) 노드, 내부(internal) 노드를 생성한 후, 이를 이용하여 다중 컬럼(multicolumn)을 구성하고 검색을 진행하는 구조이다.

또한 길이의 범위를 정하여 그에 속하는 프리픽스만 수용하는 복수의 테이블을 구성하여 검색을 진행하는 알고리즘이 제안되었다. 검색 시 제일 작은 길이의 범위를 갖는 테이블부터 검색을 진행하며 그 테이블에 검색 결과가 존재한다면 그 테이블에 해당하는 다음 홉 정보가 저장된 테이블에 접근하고, 그렇지 않다면 다음으로 긴 길이의 범위를 갖는 테이블에서 검색을 진행하여 길이가 긴 IPv6주소에서의 검색 성능을 향상 시켰다[14].

Ⅲ. 제안하는 구조

본 논문에서는 프리픽스의 길이를 레벨로 이용하여 이진 검색을 하는 구조가 IPv6 주소 검색에 적합하며, 검색 성능을 크게 향상시킬 수 있다는 점을 이용한다. 본 논문에서 제안하는 구조는 그림 2에 보여준 레벨에 따른 이진 검색 구조를 위해 생성된 노드 정보를 통합 복수-해쉉 테이블에 저장하되, 통합 블룸 필터를 사용하여 선 필터링을 수행함으로서 외부 메모리인 통합 복수-해쉉 테이블에 접근하는 횟수를 최소화하여, 검색 성능을 향상시킨 구조이다.

3.1 CRC 해쉉 함수

CRC 생성기는 길이가 다양한 입력 값을 받아 일정한 규칙을 사용하여 입력 값의 각 비트를 섞어, 모두 같은 길이의 스크램블 된 값을 생성한다. 이 구조를 이용하면 쉽게 입력 값을 섞을 수 있으며, 또한 구현이 쉽다는 장점이 있다. CRC 생성기는 다양한 길이의 입력 값을 받아 일정한 길이의 결과 값을 생성하여, 그 결과 값에서 사용자가 원하는 길이의 해쉉 색인을 마음대로 뽑을 수 있다는 장점 때문에 본 논문에서는 해쉉 색인을 구하는데 CRC 생성기를 사용하였다. 또한 다수의 프리픽스가 해쉬 테이블의 같은 위치에 지정될 수 있는데 이것을 충돌(collision)이라 한다. 충돌을 최소화하는 해쉉 색인을 구하여 효율적인 해쉉을 하는 것이 관건이며, 입력 값에 대해 균일한 출력 값을 가져야 한다[15]. CRC 생성기는 해쉉 함수 중 완전 해쉉(perfect hashing)에 가까운 성능을 낸다고 평가되고 있다[16].

그림 9에 5-비트 CRC 생성기의 예를 보였다. CRC 생성기를 사용하여 해쉉 색인을 구하는 과정은 다음과 같다. 다양한 길이의 프리픽스가 CRC 생성기에 들어가면 일정한 규칙에 의해 각 비트가 섞이게 되고, 모두 같은 길이의 스크램블 된 값이 생성된다. 이 값에서 임의로 사용자가 원하는 비트 수만큼 선택하여 색인으로 사용하게 된다. CRC내의 모든 플립-플롭(flip-flop)은 0으로 초기화되어 있으며, 인풋이 한 비트씩 들어오면서 플립-플롭의 값과 XOR 된다. 마지막 비트까지 들어오면 CRC 생성기의 동작이 멈추고 플립-플롭에 저장되어 있는 값이

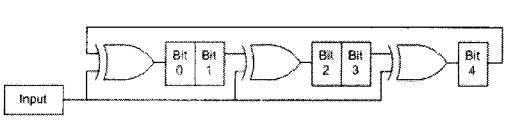


그림 9. 5비트-CRC 하드웨어 구조

CRC 코드 값이 된다.

레벨에 따른 이진 검색 구조를 해쉬 구조에 저장하기 위해서는 그림 2의 노드 정보 중 최대 길이인 5비트를 기준으로 하여, 5비트-CRC 하드웨어를 사용한다. 5비트-CRC에 프리픽스 노드(P), 마커 노드(M), BMP 선계산 노드(BMP) 값을 통과시켜 얻어낸 CRC코드는 표 2와 같다. 표 2에서 볼 수 있듯이, 다양한 길이의 노드 값들이 5비트-CRC 생성기를 거쳐 모두 길이 5의 코드값이 생성되었으며, 이 코드에서 필요한 비트를 사용자 임의로 선택하여 다양한 해쉬 색인을 얻을 수 있다.

3.2 통합 블룸 필터를 사용한 레벨에 따른 이진 검색 구조

레벨에 따른 이진 검색 구조의 노드 정보를 해쉬 구조에 저장하기에 앞서 더욱 효율적인 검색 성능을 위해 통합 블룸 필터를 사용한다. 노드 정보를 통합 블룸 필터에 프로그래밍하는 과정은 다음과 같다. N 개의 프리픽스를 저장하기 위한 블룸 필터 사이즈는 원하는 필터링 성능에 따라 조절 가능한데, 블룸 필터의 크기 $M=N'=2^{\lceil \log_2 N \rceil}$로 할 때 블룸 필터의 해쉬 색인에 필요한 비트 수는 $\lceil \log_2 N \rceil$이다. 해쉬 함수의 수(k)는 2로 정의하였다. 본 논문에서는 해쉬 색인으로 CRC 코드의 첫 $\lceil \log_2 N \rceil$비트와 마지막 $\lceil \log_2 N \rceil$ 비트를 사용하였으며, 각 노드의 해쉬 색인의 값을 표 3에 보였다.

블룸 필터를 프로그래밍하는 과정은 다음과 같다. 그림 2의 노드의 수(N)는 12이므로 블룸 필터의 사이즈 $M=N'=16$ 이다. 블룸 필터의 모든 비트-벡터의 값을 0으로 초기화 한 뒤, 각 색인의 값을 포인터로

표 2. 5비트-CRC 코드

노드 종류	노드 값	CRC 코드
M	00*	00000
M	001*	10101
P	0011*	01111
P	01*	10101
P	1*	10101
BMP	10*	11010
P	100*	01101
P	11*	01111
BMP	111*	00010
P	1110*	00001
P	11100*	10000

표 3. 해쉬 색인

노드 값	CRC 코드	해쉬 색인 1 (십진수 값)	해쉬 색인 2 (십진수 값)
00*	00000	0000(0)	0000(0)
001*	10101	1010(10)	0101(5)
0011*	01111	0111(7)	1111(15)
01*	10101	1010(10)	0101(5)
1*	10101	1010(10)	0101(5)
10*	11010	1101(13)	1010(10)
100*	01101	0110(6)	1101(13)
11*	01111	0111(7)	1111(15)
111*	00010	0001(1)	0010(2)
1110*	00001	0000(0)	0001(1)
11100*	10000	1000(8)	0000(0)

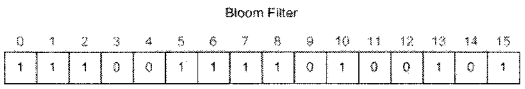


그림 10. 프로그래밍 된 블룸 필터(사이즈=N')

사용하여 해당하는 비트-벡터의 값을 1로 프로그래밍 한다. 그림 2의 모든 노드 정보를 블룸 필터 사이즈 N'에 프로그래밍 한 결과는 그림 10과 같다.

3.3 통합 복수-해쉬 테이블을 이용한 레벨에 따른 이진 검색 구조

본 논문에서는 길이를 이용한 이진 검색 구조의 노드 정보를 통합 복수-해쉬 테이블에 저장하여 검색 성능을 향상시키고, 메모리 요구량에 있어서도 좋은 성능을 보이는 구조를 제안한다. 하나의 테이블당 버켓의 수를 $2^{\lceil \log_2 N \rceil}=N'$, 하나의 버켓 당 엔트리의 수를 2로 갖는다고 가정하였다.

통합 복수 해쉬 테이블에 프리픽스를 저장하는 과정을 보면 다음과 같다. 표 3의 해쉬 색인의 값이 해쉬 테이블의 메모리를 가리키는 포인터가 되며, 해쉬 색인 1과 해쉬 색인 2를 사용하여 두 개의 테이블에 동시에 접근한다. 접근한 두 개의 버켓에서 로드의 개수를 비교하여 로드의 갯수가 적은 쪽에 프리픽스를 저장한다. 두 개의 버켓의 로드의 수가 같은 경우 임의로 첫번째 테이블에 저장되는 것을 가정하였다. 만일 두 개의 버켓이 모두 차 있다면 오버플로우 테이블에 프리픽스를 저장한다.

통합 복수 해쉬 테이블의 크기에 따라 오버플로우의 개수가 조절 가능하며, 적절한 크기의 메모리를 사용하여 오버플로우 되는 프리픽스의 개수를 일정

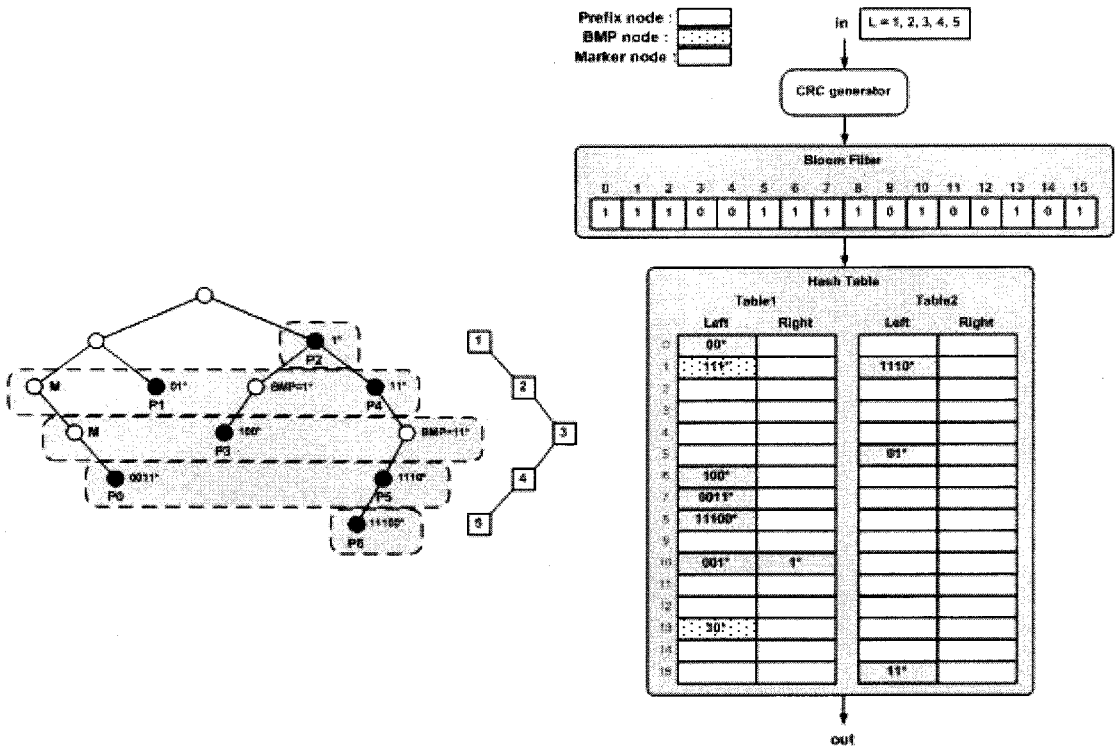


그림 11. 제안하는 구조

수 이하로 제한 할 수 있으며, 오버플로우 테이블은 캐쉬나 작은 크기의 TCAM으로 구현 가능하다.

3.4 검색 과정

본 논문에서 제안하는 구조는 길이를 이용한 이진 검색이므로, 최장길이 일치 프리픽스를 찾기 위해서는 입력값에 대하여 트라이 레벨에 따라 이진 검색을 진행한다. 그림 11에 본 논문에서 제안하는 레벨에 따른 이진 검색 구조를 보였다. 그림 11의 트라이에는 레벨에 다른 이진 검색을 위한 트라이 구조에서 저장되어야 하는 노드들과 접근 레벨의 순서를 표시하였다. CRC 생성기 및 통합 블룸 필터는 IPv6 주소 검색을 위하여 설계된 칩의 내부에 존재하고, 통합 복수 해쉬 테이블은 칩의 외부에 존재하는 것을 가정하여, 블룸 필터를 사용하여 칩 외부 메모리 접근 회수를 최소화하는 것을 목표로 한다. 이는 향후 저장하여야 할 IPv6의 프리픽스 정보가 급격히 증가하여 커다란 메모리를 요구하는 경우에도 실제 노드 정보를 저장하는 메모리를 칩 외부에 둠으로서 쉽게 적용할 수 있다는 장점이 있다.

그림 11에서 알 수 있듯이 레벨 3에 해당하는 노드에 가장 먼저 접근하며, 만일 일치하는 노드(마커 노드, 프리픽스 노드 혹은 BMP 노드)가 없다면 레벨 2로, 일치하는 노드(마커 노드, 프리픽스 노드 혹은 BMP 노드)가 있다면 레벨 4으로 검색을 진행한다. 레벨에 따른 이진 검색의 마지막 레벨(레벨 1 혹은 레벨 5)까지 검색을 진행 후, 일치하는 결과 중 최대 길이의 값이 BMP가 되며 그 포워딩 정보를 출력한다.

검색 과정을 그림 12에 보였으며, 그림 12로 검색 과정을 설명하면 다음과 같다. 먼저 입력 값을 현재

```
Search(input)
{
  do {
     in = Cut_input(input, L);
     code = CRC(in, 128);
     ind1 = first ⌈log2N⌉ bits of code;
     ind2 = last ⌈log2N⌉ bits of code;
     if( (All length Bloom Filter(ind1)==1)
        && (All length Bloom Filter(ind2)==1) ) {
        if((All length Multi-hashing Tbl1(ind1)==in)
          || (All length Multi-hashing Tbl2(ind2)==in) ){
          BMP=in;
          if(L is final access level) return;
          else L=next access level of the long length;
        }
        else L=next access level of the short length;
     }
     else L=next access level of the short length;
  }while (L != final access level);
}
```

그림 12. 검색 과정 pseudo-code

접근 길이에 맞추어 자른 후, CRC-생성기에서 코드를 구한다. 코드 값에서 구해진 해쉉 색인을 사용하여 블룸 필터에서 쿼링을 진행한다. 두 색인 값의 위치에 해당하는 비트-벡터의 값 중 하나라도 0이 존재하면, 이 길이는 일치하는 노드가 없는 길이이므로 해쉉 테이블에서의 검색을 진행할 필요가 없다. 그러므로 바로 길이가 짧은 쪽의 다음 레벨로 검색을 진행한다. 만일 비트-벡터의 값이 모두 1인 경우에는 이 길이의 노드가 해쉉 테이블에 존재할 가능성이 있으므로, 인풋은 블룸 필터를 통과하여, 두 색인을 이용하여 해쉉 테이블에 인풋이 실제로 존재하는지 검색을 진행한다. 그림 11의 Table1과 Table2는 별도의 SRAM에 저장되어 있는 것을 가정하여, 검색 시 병렬적으로 메모리에 접근하여 검색을 진행한다. 해쉉 색인 1의 값으로 Table1의 버켓을 검색하고, 해쉉 색인 2의 값으로 Table2의 버켓을 검색하여 인풋과 일치하는 노드가 저장되어 있는지 확인한다. 검색 결과 일치하는 노드가 있다면 이는 지금까지의 BMP이므로 이 결과를 기억한 후 더 긴 레벨로 검색을 진행한다. 만일 검색 도중 해쉉 테이블에 일치하는 노드가 없다면 오버플로우 테이블에서 검색을 진행하여 이때 일치가 있다면 길이가 긴 쪽의 다음 레벨로 검색을 진행하며, 오버플로우 테이블에서도 일치가 없다면, 이는 블룸 필터에서 거짓 양성을 만들어 낸 경우로서, 길이가 짧은 쪽의 다음 레벨로 검색을 진행한다.

예를 들어 입력된 주소가 00110이라고 가정하면, 그림 11에서 볼 수 있듯이 첫 접근 길이는 3이므로 입력된 주소의 처음 세 비트 001을 CRC 생성기에 입력한다. 5비트-CRC의 코드는 10101이며 블룸 필터와 해쉉 테이블의 색인은 10과 5가 된다. 통합 블룸 필터에 이 색인 값으로 쿼링을 진행하면 블룸 필터의 10번째와 5번째 비트-벡터 값은 모두 1이므로 양성인 경우이고, 통합 복수 해쉉 테이블에 이 노드가 존재할 가능성을 의미한다. 따라서 색인 값으로 통합 복수 해쉉 테이블에서 검색을 진행한다. Table1의 Left에서 입력 값과 일치하는 노드 정보가 존재한다. 001* 노드는 프리픽스가 저장되어 있지 않지만, 그 노드보다 길이가 긴 쪽의 노드에 프리픽스 노드가 존재한다는 마커 노드이므로 다음 긴 레벨로 검색을 진행한다. 다음 검색 길이는 4이므로 입력 값의 처음 네 비트 0011이 CRC 생성기를 통과한 코드 값은 01111이며 색인은 7과 15가 된다. 이 값은 블룸 필터를 통과하게 되며, 통합 복수 해쉉 테이블의 Table1 7번째 엔트리와 Table2 15번째 엔트리에서 검색을 진행한다. Table1의 Left에서 0011* 프리픽스 노드가 존재하므로 현재까지 BMP는 0011*이 된다. 일치하는 노드가 존재하므로 다음 긴 레벨인 5에서 검색을 진행한다. 입력 주소 00110을 CRC 생성기를 통과시키면 10111인 코드 값이 나온다. 색인은 11과 7이며 통합 블룸 필터에서 11번째 비트-벡터 값이 0이므로 이 값은 통합 블룸 필터를 통과하지 못하며, 길이 5는 마지막 접근 길이이므로 검색은 현재까지의 BMP인 0011*이 출력되며 검색이 종료된다. 이 경우 통합 블룸 필터가 존재하지 않는다면 3번의 통합 복수 해쉉 테이블로의 접근이 필요하지만 통합 블룸 필터가 멤버가 아닌 값을 필터링 해 주므로 2번의 통합 복수 해쉉 테이블로의 접근으로 검색을 마칠 수 있다. 이와 같이 통합 블룸 필터는 불필요한 통합 복수 해쉉 테이블로의 접근을 막아주어 효율적인 검색을 제공한다.

3.5 업데이트

제안하는 구조에 새로운 노드 정보를 추가하거나, 기존의 노드 정보를 제거하는 과정을 살펴보면 다음과 같다. 새로 추가할 노드 정보를 CRC 하드웨어에 넣어 코드를 얻고, 그 코드에서 해쉉 색인을 구한다. 두 해쉉 색인으로 통합 블룸 필터를 프로그래밍하며, 통합 복수 해쉉 테이블에 접근하여 해쉉 색인 값에 위치하는 곳의 로드를 확인 후, 로드가 적은 곳에 노드 정보를 삽입한다. 만일 통합 복수 해쉉 테이블의 로드가 모두 차 있다면 오버플로우 테이블에 노드 정보가 저장된다.

노드 정보의 삭제는 삭제할 정보의 CRC 아웃풋을 구한 뒤, 이 값에서 해쉉 색인을 구한다. 이 두 색인값으로 통합 복수 해쉉 테이블에서 색인에 해당하는 테이블 엔트리를 검색하며, 삭제할 노드 정보가 발견되면 그 정보를 삭제한다. 삭제의 경우 추가와는 달리 통합 블룸 필터에서의 삭제는 이루어지지 않는다. 따라서 검색 시 삭제하지 않은 노드 정보로 인해 통합 블룸 필터의 거짓 양성(false positive)은 증가할 것이지만 통합 복수 해쉉 테이블에서의 검색은 정확하게 이루어진다.

만일 통합 블룸 필터에서의 삽입, 삭제 동작이 매우 많이 반복되는 경우에는 블룸 필터의 거짓 양성의 확률이 매우 증가될 수 있다. 업데이트 빈도가 높은 라우터에서는 통합 블룸 필터의 각 비트-벡터에 해당하는 구성 성분의 수를 카운트하여, 이 정보를 저장하는 공간을 추가한 카운팅 블룸 필터를 사용할 수 있다. 즉, 블룸 필터에 정보가 삽입되었을

때는 해당 비트의 카운트 정보의 값을 1 증가시키며, 삭제가 되었을 때는 1 감소시킨다. 만일 카운트 값이 0이 되었을 경우에는 블룸 필터의 비트-벡터 값을 1에서 0으로 셋팅하여 거짓 양성 비율의 증가를 막는다.

또한 본 논문에서 제안하는 구조는 프리픽스의 각 비트를 스크램블하여 프리픽스집합의 엔트리 수에 해당하는 비트를 취하여 해싱을 하므로 프리픽스의 길이와 직접적인 관련이 없다. 따라서 IPv4 또는 IPv6의 프리픽스로 제안하는 알고리즘을 구성하여도 큰 차이점이 없으나, 프리픽스 집합의 엔트리 수에 영향을 받으므로 프리픽스 개수(N)의 $2^{\lceil \log_2 N \rceil}$에 해당하는 엔트리 구조를 갖는 해쉬 테이블을 사용하여야 한다. 즉, 업데이트로 인하여 프리픽스 집합의 엔트리 수가 증가하였을 때, 해싱을 하여 메인 테이블에 저장하는 것을 원칙으로 하며, 충돌이 발생하였을 시에는 오버플로우 테이블에 저장한다. 만일 최악의 경우 오버플로우 테이블에 저장되는 프리픽스의 수가 많아 비용 증가가 심할 경우 최종 엔트리 개수(N)의 $2^{\lceil \log_2 N \rceil}$에 해당하는 해쉬 테이블을 다시 구성하는 방법을 적용하여야 한다.

Ⅳ. 제안하는 구조의 성능 평가

본 논문에서는 실제 백 본 라우터에서 사용된 1069개의 IPv6 라우팅 데이터[17]와 1069개 중 100개와 500개를 랜덤으로 뽑아 실험하였다. 실제 IPv6 라우팅 테이블에 존재하는 프리픽스들은 최대 길이가 128비트이므로 시뮬레이션에서는 64비트-CRC 2개를 연결하여 사용하였다.

표 4는 각 프리픽스 집합에서 통합 블룸 필터와 통합 복수 해싱 테이블의 크기에 따라 발생하는 오버플로우 개수를 보이고 있다. 표 4에서 노드의 개수는 프리픽스 집합에서의 프리픽스 개수보다 많은데, 이것은 레벨에 따른 이진 검색 구조에서 생성되는 프리픽스 노드, 마커 노드, BMP노드의 정보를 모두 포함해야 하기 때문이다. 통합 복수 해싱 테이블의 버켓의 개수를 $2^{\lfloor \log_2 N \rfloor}$와 $2^{\lceil \log_2 N \rceil}$으로 하였을 때의 오버플로우 수와 메모리 사용량을 나타내었다. 각 프리픽스 집합에서 거의 모든 노드 정보가 통합 복수 해싱 테이블에 저장되며, 통합 복수 해싱 테이블의 엔트리 수와 오버플로우 수는 트레이드-오프 관계에 있다는 것을 알 수 있다.

표 4. 제안하는 해싱 테이블의 엔트리 수와 오버플로우 수, 메모리 사용량

프리픽스 개수 (N)	노드 개수	N'	Table Size	오버플로우 개수	메모리 사용량 (kbyte)
100	160	128	27	0	9.13
		256	28	0	18.25
500	1422	1024	210	0	73
		2048	211	0	146
1069	3383	2048	211	2	146 + 2-entry TCAM
		4096	212	0	292

통합 복수 해싱 테이블의 엔트리 구조와 블룸 필터와 통합 복수 해싱 테이블을 위한 메모리 사용량을 계산하는 방법을 그림 13에 보였다.

현재 네트워크 상에 존재하는 대부분의 라우터는 빠른 인터넷 주소 검색 성능을 요구하므로 메모리 접근 횟수가 메모리 요구량보다 우선적으로 요구된다. 따라서 본 논문에서 제안하는 구조에서 오버플로우를 줄이기 위하여 통합 복수 해싱 테이블의 크기를 증가시키면 메모리 요구량이 증가하지만, 타 구조에 비하여 월등히 빠른 검색 성능을 보인다는 것을 알 수 있다.

복수 해싱은 완전 해싱으로 간주할 만큼의 해싱 성능을 보인다는 것이 보고되어 왔으며, IPv6의 라우팅 테이블의 경우에도 오버플로우의 급격한 증가 우려가 매우 적다.

표 5와 그림 14에 제안하는 구조의 성능을 보였다. 통합 복수 해싱 테이블의 경우 버켓의 수가 $N'=2^{\lceil \log_2 N \rceil}$일 때이며, 통합 블룸 필터의 경우 크기를 N', $2N'$, $4N'$, $8N'$, $16N'$으로 증가시키며 실험하였다. 표 5에서 BT_{max}는 블룸 필터의 최대 접근(probe) 회수를 의미하여, BT_{avg}는 블룸 필터의 평균

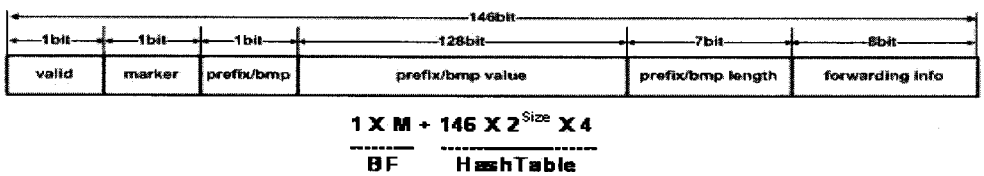


그림 13. 테이블의 엔트리 구조

표 5. 제안하는 구조의 성능

				통합 블룸 필터			통합 복수해쉬 테이블			해쉬 테이블 접근율(%)
프리픽스 개수 (N)	노드 개수	N'	접근 레벨 수	크기	BT_{max}	BT_{avg}	크기	HT_{max}	HT_{avg}	
100	160	256	11	N'	4	3.09	28	4	2.10	67.96
				$2N'$				4	1.60	51.78
				$4N'$				4	1.40	45.31
				$8N'$				4	1.30	42.07
				$16N'$				4	1.28	41.42
500	1422	2048	17	N'	5	4.77	211	5	4.05	84.91
				$2N'$				5	3.38	70.86
				$4N'$				5	3.07	64.36
				$8N'$				5	2.97	62.26
				$16N'$				5	2.93	61.43
1069	3383	4096	23	N'	5	4.20	212	5	3.34	79.52
				$2N'$				5	2.40	57.14
				$4N'$				5	1.97	46.90
				$8N'$				5	1.67	39.76
				$16N'$				5	1.64	39.05

표 6. 제안하는 구조와 해쉬 기반 알고리즘의 오버플로우 수 비교

프리픽스 개수 (N)	# of node	Proposed	병렬 해싱[6]	병렬 복수 해싱[8]
100	160	0	39	0
500	1422	0	356	0
1069	3383	0	798	0

접근 회수를 의미한다. 또한 HT_{max}는 통합 해싱 테이블의 최대 접근 (probe) 회수를 의미하여, HT_{avg}는 통합 해싱 테이블의 평균 접근 회수를 의미한다. 해쉬 테이블 접근 비율(%)은 주소 검색시 통합 블룸 필터 접근 회수대비 통합 복수 해싱 테이블 접근 회수의 비율이다. 통합 블룸 필터의 크기가 증가할수록 해쉬 테이블 접근 비율이 감소하며, 이것은 통합 블룸 필터의 필터링 능력이 증가하면서 거짓 양성(false positive)이 감소하는 것을 뜻한다. 따라서 통합 블룸 필터의 크기가 증가할수록 통합 복수 해싱 테이블로

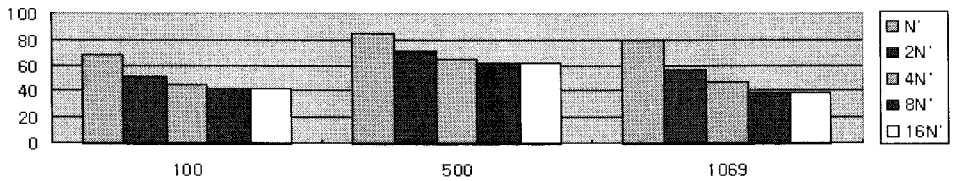


그림 14. 제안하는 구조의 해싱 테이블 접근율 (%)

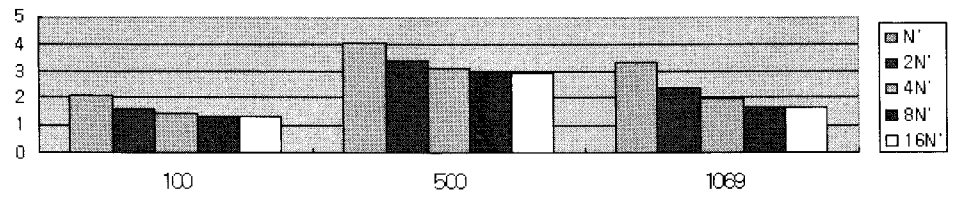


그림 15. 통합 블룸 필터의 크기에 따른 통합 복수 해싱 테이블의 평균 접근횟수(HT_{avg})

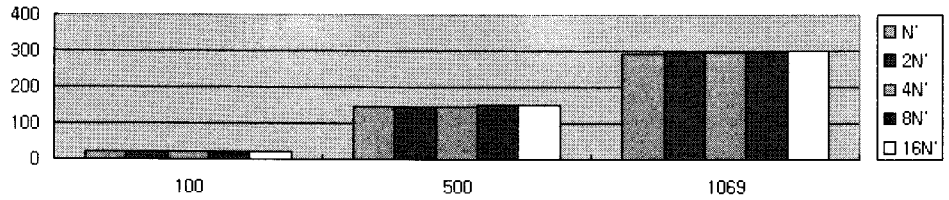


그림 16. 통합 블룸 필터의 크기에 따른 메모리 요구량 (kbyte)

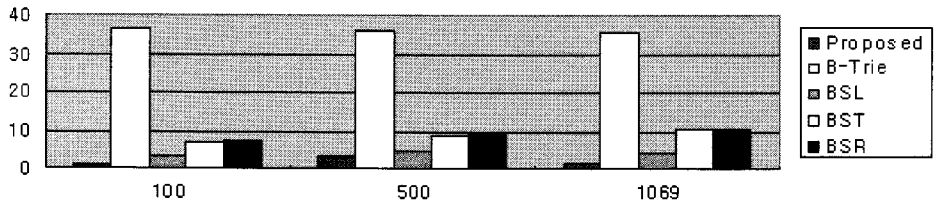


그림 17. 기존의 이진 검색 알고리즘과 평균 메모리 접근 횟수

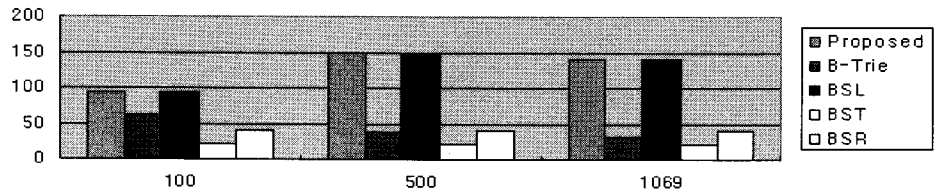


그림 18. 기존의 이진 검색 알고리즘과 프리픽스 당 메모리 요구량(byte/prefix)

의 불필요한 메모리 접근이 감소함을 뜻한다.

그림 15, 그림 16에서 통합 블룸 필터의 크기가 증가할 때, 제안하는 구조의 평균 메모리 접근횟수와 메모리 요구량을 나타내었다. 통합 블룸 필터의 크기가 증가하면 필터링 능력이 증가하여 통합 복수 해쉬 테이블로의 접근이 감소됨을 알 수 있다. 또한 그림 16으로부터 통합 블룸 필터의 크기는 제안하는 구조의 전체 메모리 요구량에 거의 영향을 미치지 않는 다는 것을 알 수 있다.

그림 17에서는 제안하는 구조의 버켓의 수를 $N'=2^{\lfloor \log_2 N \rfloor}$, 블룸 필터의 크기를 $16N'$으로 하여 다른 알고리즘과 평균 메모리 접근 횟수를 비교하였다. 여기서 기존의 레벨에 따른 이진 검색 구조[3]는 제안하는 구조에서 설명한 통합 복수 해쉬 테이블만을 사용하여 구현한 경우로 가정하였다. 이 구조에서는 프리픽스가 존재하는 모든 레벨에 대하여 외부 메모리를 접근하여야 하므로, 제안하는 구조에서의 통합 블룸 필터로의 접근 횟수로 생각할 수 있다. 제안하는 구조는 적은 메모리를 요구하는 통합 블룸 필터 사용으로 인하여 기존의 레벨에 따른 이진 검색 구조에 비하여 검색 성능을 더욱 향상시킨 것으로 판단된다. 제안하는 구조가 타 알고리즘에 비해 최대 29배 성능이 우수함을 볼 수 있어, 본 논문에서 제안하는 구조가 IPv6 주소 체계에 더욱 적합함을 알 수 있다.

그림 18에서는 제안하는 구조의 버켓의 수가 $N'=2^{\lfloor \log_2 N \rfloor}$, 블룸 필터의 크기가 $16N'$일 때 메모리 요구량을 비교하였다. 프리픽스 당 메모리 요구량의 경우 타 알고리즘에 비해 제안하는 구조가 최대 7배까지 많은 메모리를 요구하는 것을 알 수 있으나, 실제 라우터의 경우 입력 패킷을 포워딩하는 속도가 링크 속도에 미치지 못한다는 점을 감안할 때 평균 메모리 접근 횟수(속도)가 메모리 요구량 보다 더 중요한 평가 항목이 될 것이다.

표 6에서는 $N'=2^{\lceil \log_2 N \rceil}$일 때, 제안하는 구조와 해쉬 기반 알고리즘의 오버플로우 수를 비교하였다. 제안하는 구조와 병렬복수 해쉬 구조는 모든 프리픽스가 주-테이블에 저장된다는 것을 알 수 있으며, 병렬 해쉬 구조의 경우 1069개의 프리픽스 집합에서 798개의 프리픽스가 주-테이블에 저장되지 못하고 보조-테이블에 저장된다는 것을 알 수 있다.

그림 19에서는 본 논문에서 제안하는 구조의 버

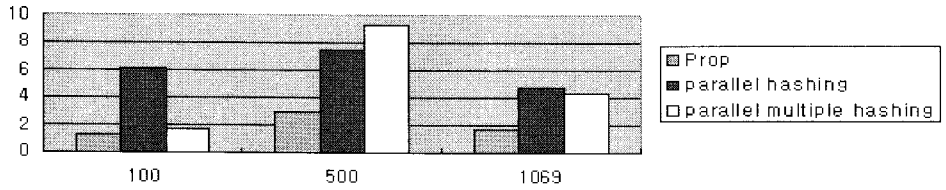


그림 19. 제안하는 구조와 해쉬 기반 알고리즘의 평균 메모리 접근 횟수 비교

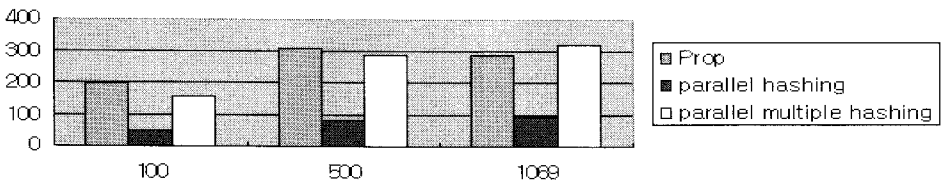


그림 20. 제안하는 구조와 해쉬 기반 알고리즘의 프리픽스 당 메모리 요구량비교 (byte)

켓의 수가 $N'=2^{\lceil \log_2 N \rceil}$, 블룸 필터의 크기가 $16N'$일 때와 해쉬 기반 알고리즘의 검색 성능을 평균 메모리 접근 횟수로 비교하였다. 병렬 해싱과 병렬 복수 해싱의 경우 해싱 테이블의 버켓의 수를 $2^{\lceil \log_2 N \rceil}$로 하여 비교하였다. 또한 동일한 조건에서 성능을 비교하기 위하여, 해쉬 기반 알고리즘의 경우 병렬처리 방식이 아닌 길이에 따른 이진 검색으로 검색 성능을 측정하였다. 즉, 그림 2에서 생성된 프리픽스 노드, 마커 노드, BMP 노드 정보를 각 구조에 저장하여 각 레벨을 이용하여 이진 검색 방식으로 접근 레벨을 결정하였다. 그 결과 제안하는 구조가 타 구조에 비하여 1.6-5.7배 빠른 검색 성능을 보였다.

그림 20에서는 각 구조의 프리픽스 당 메모리 요구량을 비교하였다. 제안하는 구조의 버켓의 수는 $N'=2^{\lceil \log_2 N \rceil}$, 블룸 필터의 크기는 $16N'$으로 해쉬 기반 알고리즘의 경우 버켓의 수를 $2^{\lceil \log_2 N \rceil}$으로 하여 비교하였다. 그 결과 제안하는 구조는 병렬 해싱보다는 많은 메모리 요구량을, 병렬 복수 해싱보다는 적은 메모리를 요구한다는 것을 알 수 있으며, 병렬 해싱의 경우 타 구조에 비하여 많은 프리픽스가 보조 테이블에 저장되어 이진 검색을 통하여 검색되므로, 검색 성능이 저하된다는 점을 감안할 때, 제안하는 구조와 병렬 복수 해싱 구조가 더 IPv6의 주소 검색 구조로서 더 우수하다고 판단된다.

V. 결 론

본 논문에서는 길이가 긴 IPv6 주소 검색을 위한 프리픽스 레벨에 따른 이진 검색 구조를 제안하였다. 본 논문에서 제안된 구조는 레벨에 따른 이진 검색을 수행하되, 블룸 필터를 이용하여 외부 메모리로의 접근 회수를 감소시킨 구조로서, 1000여개의 IPv6 프리픽스 정보를 갖는 라우팅 테이블에 대하여 1~3번의 메모리 접근만으로 최장 길이 일치 프리픽스를 검색 할 수 있는 효율적인 구조이다

참 고 문 헌

[1] M. A. Ruiz-Sanchex, E. W. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms", IEEE Network, pp.8-23, March/April 2001.

[2] H. J. Chao, "Next generation routers," Proc. of the IEEE, vol.90, no.9, pp.1518-1558, Sep. 2002.

[3] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups", in Proc. ACM SIGCOMM Conf., Cannes, France, pp.25-35, 1997.

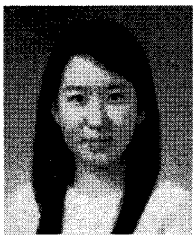
[4] N. Yazdani and P. S. Min, "Fast and scalable schemes for the IP address lookup problem," Proc. IEEE HPSR2000, pp. 83-92.

[5] Priyank Warkhede, Subhash Suri, and George Varghese. "Multiway range trees: scalable IP lookup with fast updates," Computer Networks, pp. 289-303, 2004.

[6] Hyesook Lim, Ji-Hyun Seo, and Yeo-Jin Jung, "High speed IP address lookup architecture using hashing," IEEE Communications Letters,

vol.7, no. 10, pp. 502-504, October 2003.
[7] Andrei Broder and Michael Mitzenmacher, "Using multiple hash functions to improve IP lookups," Proc. IEEE Infocom 2001, vol.3, pp. 1454-1463.
[8] Hyesook Lim and Yeojin Jung, "A parallel multiple hashing architecture for IP address lookup," Proc. HPSR 2004, pp. 91- 95.
[9] 박경혜, 임혜숙, "IP 주소 검색에서 블룸 필터를 사용한 다중 해싱 구조," 정보과학회논문지: 데이타베이스, 제 36권 제 2호, pp. 84-98, 2009.4.
[10] Sarang Dharmapurikar, Praveen Krishamurthy, and David E. Taylor, "Longest prefix matching using bloom filters," IEEE/ACM Transactions on Networking, vol. 14, no. 2, pp. 397-409, April 2006.
[11] Jahangir Hasan, Srihari Cadambi, Venkatta Jakkula, and Srimat Chakradhar, "Chisel: A storage-efficient, collision-free hash-based network processing Architecture," Proc. ISCA 2006. pp. 203-215.
[12] Alagukonar Ganapathy Alagupriya, "Packet Classification Algorithms Using Bloom Filters", 이화여자대학교 대학원 2008년도 석사학위 청구 논문.
[13] Yiu Keung Li and Derek Pao, "Comparative Studies of Address Lookup Algorithms for IPv6," Proc. ICACT 2006, pp.285-290, Feb 2006.
[14] S. M. Yong and H. T. Ewe, "Robust Routing Table Design for IPv6 Lookup," Proc. ICITA'05, 2005.
[15] Rich Seifert, "The Switch book," Wiley, 2000.
[16] Raj Jain, "Comparison of Hashing Schemes for Address Lookup in Computer Networks," in IEEE Transactions on Communications, 1989.
[17] IPv6 Operational Report. http://net-stats.ipv6.tilab.com/bgp/bgp-table-snapshot.txt/, June 2008.

박 경 혜 (Kyong Hye Park) 준회원



2007년 2월 이화여자대학교 정보통신학과 학사
2009년 2월 이화여자대학교 전자정보통신공학과 석사
<관심분야> Router나 switch 등의 Network 관련 SoC 설계, TCP/IP 관련 하드웨어 설계

임 혜 숙 (HyeSook Lim) 종신회원

1986년 2월 서울대학교 제어계측공학과 학사
1986년 8월~1989년 2월 삼성 휴렛팩커드 연구원
1991년 2월 서울대학교 제어계측공학과 석사
1996년 12월 The University of Texas at Austin, Electrical and Computer Engineering, Ph.D.
1996년 11월~2000년 7월 Lucent Technologies, Bell Labs, Member of Technical Staff
2000년 7월~2002년 2월 Cisco Systems, Hardware Engineer
2002년 3월~현재 이화여자대학교 공과대학 전자공학과 부교수
<관심분야> Rourer나 switch 등의 Network 관련 SoC 설계, TCP/IP 관련 하드웨어 설계