IP 주소 검색에서 블룸 필터를 사용한 다중 해싱 구조

(Multiple Hashing Architecture using Bloom Filter for IP Address Lookup)

박 경 혜 † 임 혜 숙 ††
(Kyong Hye Park) (Hyesook Lim)

요 약 라우터의 포워딩 성능을 향상시키기 위해 많은 IP 주소 검색 알고리즘들이 연구되어 오고 있다. 기존에 제안된 블룸 필터를 이용한 IP 주소 검색 구조는 프리픽스 길이별로 블룸 필터 및 해시 테이블을 따로 가지고 있어 구현이 복잡하며, 프리픽스 길이 별 해시 테이블의 개수를 줄이기 위하여 프리픽스의 복사가 불가피한 단점을 지닌다. 멀티 다중 해시 테이블을 이용한 병렬 해싱 구조는 프리픽스의 길이별로 해시 테이블을 구성하고, 다중 해싱 함수를 사용하여 모든 프리픽스 길이에 대하여 병렬 검색하는 구조로서 검색 성능이 뛰어나나 병렬 검색 구조이므로 구현이 또한 복잡하다. 본 논문에서는 단일 블룸 필터에 다양한 길이의 프리픽스를 모두 저장하는 통합 블룸 필터와 단일 테이블에 모든 길이의 프리픽스를 모두 저장하는 통합 다중-해시 테이블을 사용하여 구현이 간단하면서도 검색성능이 뛰어난 새로운 IP 주소 검색 구조를 제안한다. 실제 백본 라우터에서 쓰이는 데이타를 이용하여 시뮬레이션을 수행한 결과 15000~220000개의 엔트리를 갖는 라우팅 테이블에 대하여 평균 1.04-1.17번의 메모리 접근으로 IP 주소 검색이 가능함을 보였다.

키워드 : IP 주소 검색, 최장 길이 일치 프리픽스, 해싱, 블룸 필터

Abstract Various algorithms and architectures for IP address lookup have been studied to improve forwarding performance in the Internet routers. Previous IP address lookup architecture using Bloom filter requires a separate Bloom filter as well as a separate hash table in each prefix length, and hence it is not efficient in implementation complexity. To reduce the number of hash tables, it applies controlled prefix expansion, but prefix duplication is inevitable in the controlled prefix expansion. Previous parallel multiple-hashing architecture shows very good search performance since it performs parallel search on tables constructed in each prefix lengh. However, it also has high implementation complexity because of the parallel search structure. In this paper, we propose a new IP address lookup architecture using all-length Bloom filter and all-length multiple hash table, in which various length prefixes are accomodated in a single Bloom filter and a single multiple hash table. Hence the proposed architecture is very good in terms of implementation complexity as well as search performance. Simulation results using actual backbone routing tables which have 15000~220000 prefixes show that the proposed architecture requires 1.04-1.17 memory accesses in average for an IP address lookup.

Key words : IP address lookup, longest matching prefix, Hashing, Bloom Filter

· This research was supported by LG Yonam Foundation under the overseas research professor program and the MIC(Ministry of Information and Communications) under a HNRC-ITRC support program supervised by IITA(Institute of Information Technology Assessment).

† 학생회원 : 이화여자대학교 전자정보통신공학과
khpark09@ewhain.net
†† 정 회 원 : 이화여자대학교 전자정보통신공학과 교수
hlim@ewha.ac.kr

논문접수 : 2008년 6월 25일
심사완료 : 2009년 1월 20일

정보과학회논문지: 데이타베이스 제36권 제2호(2009.4)

1. 서 론

라우터에 입력된 패킷을 선 속도로 포워딩하기 위해서는 보다 효율적인 IP 주소 검색이 필요하다. IP 주소 검색은 입력된 패킷의 네트워크 부분과 라우터 내에 저장된 테이블의 엔트리 중 일치하는 엔트리를 찾아 패킷의 출력포트를 정해주는 작업이다. IP 주소 검색에서는 입력 패킷의 목적지 주소의 네트워크 부분인 프리픽스 길이에 대한 정보를 미리 알고 있지 않으므로 최장길이 일치(longest prefix match) 방법을 사용하여야 하는데, 이는 입력된 패킷의 목적지 주소와 일치하는 길이가 다양한 프리픽스 중 가장 길게 일치하는 프리픽스 혹은 최적으로 일치하는 프리픽스(best matching prefix: BMP)를 찾아내는 것을 뜻한다[1,2]. IP 주소 검색과정에서 가장 중요한 것은 검색 성능으로서 BMP를 정하기 위해 소요되는 메모리 접근 횟수가 그 관건이 된다. 이는 IP 주소 검색을 위한 여러가지 과정중 메모리를 읽고 쓰는데 소요되는 시간이 다른 과정에 비하여 현저히 길기 때문이다. 다음으로 중요한 것은 라우팅 테이블을 저장하기 위한 메모리의 크기인데, 많은 프리픽스를 효율적으로 저장할 수 있어야하며, 이는 IP 주소 검색 알고리즘의 데이타 구조에 의존한다. 또한 프리픽스를 쉽게 추가 또는 삭제할 수 있는 테이블 갱신의 용이성도 중요한 요소이며, IPv4에서 IPv6로 확장성도 요구된다.

본 논문에서는 단일 블룸 필터 안에 다양한 길이의 프리픽스를 프로그래밍하는 통합 블룸 필터(Bloom filter)와 통합 다중-해시(multiple hash) 테이블을 결합한 효율적인 IP 주소 검색 구조를 제안한다. 제안하는 통합 블룸 필터는 다양한 길이의 프리픽스를 모두 수용할 수 있는 단일 블룸 필터로서 일치 가능성이 없는 프리픽스 길이를 추려내는 역할을 한다. 제안하는 통합 다중-해시 테이블 또한 단일 테이블에 다양한 길이의 프리픽스를 모두 수용하는 테이블로서, 완전해싱(perfect hashing)함수를 사용하지 않고도 다중 해싱(multiple hashing)을 사용함에 의하여 해싱의 충돌 문제를 해결하는 구조이다. 본 논문의 구성은 다음과 같다. 먼저 2장에서는 기존의 IP 주소 검색 방법들에 대해 소개한다. 3장에서는 본 논문에서 제안하는 구조에 대해 설명하며, 4장에서 제안하는 방식의 성능과 다른 구조와의 성능 비교를 보인다. 마지막으로 5장에서 결론을 맺는다.

2. 관련 연구

본 논문에서 제안하는 알고리즘과 관련된 구조로는 트리 구조를 기반으로 하는 알고리즘과, 해싱을 기반으로 하는 알고리즘, 블룸 필터에 기반한 알고리즘 등이 있다. 2장에서는 표 1에 보인 프리픽스 집합을 예로하여 각 알고리즘에 대해 간단히 설명한다.

표 1 프리픽스 집합

	Prefix
P0	001*
P1	0010*
P2	101*
P3	11111*
P4	111110*
P5	10001111*
P6	11100010*

2.1 트리 구조를 기반으로 하는 알고리즘

2.1.1 이진 트라이(B-trie)

표 1의 예로 이진 트라이를 구성해 보면 그림 1과 같다. 이진 트라이[2]는 프리픽스의 값에 따라 해당하는 트라이의 노드에 프리픽스 정보를 저장하는 방법이며, 프리픽스를 찾기 위해 거치는 경로로 프리픽스 값을 알 수 있는 장점이 있다. 따라서 프리픽스 값 자체를 따로 저장할 필요는 없으나, 그 경로에 있는 모든 노드들에 프리픽스들이 저장되는 것이 아니므로 타 알고리즘에 비해 빈 노드가 많아 메모리 요구량이 증가되는 단점이 있다. 또한 메모리 접근 횟수가 프리픽스의 최대 길이까지 커질 수 있으며, 프리픽스의 분포가 고르지 않으면 불균형 트리가 구성되어 메모리 접근 횟수가 일정치 않다는 단점이 있다[3,4]. 검색 진행 과정은 최상위 노드인 루트 노드부터 시작하며 입력 패킷의 목적지 주소의 최상위 비트(most significant bit: MSB)부터 한 비트씩 검사하여, 그 값이 0이면 왼쪽 자식 노드로 이동하고 1이면 오른쪽 자식 노드로 이동하면서 진행하되 프리픽스 노드를 만나면 노드에 저장된 출력 포트를 기억한다. 이진 트라이의 잎(leaf)에 도달하여 더 이상 진행할 노드가 없을 때 기억한 출력 포트가 최종 검색 결과가 된다.

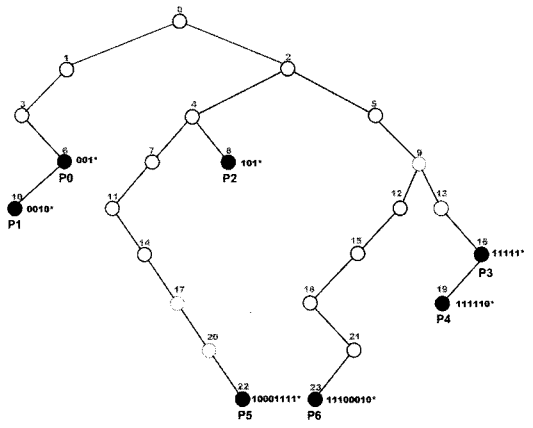

그림 1 이진 트라이

2.1.2 이진검색 트리(Binary Search Tree, BST)

이진검색 트리[5]는 다양한 길이를 갖는 프리픽스들의 크기 비교를 가능하게 하는 새로운 정의를 제시하고, 이에 따라 프리픽스를 크기별로 정렬 한 후 이진검색 트리를 구성하는 방식이다. 먼저 프리픽스 크기 비교에 관한 정의를 살펴보면, 비교하는 두 프리픽스 길이가 같은 경우는 서로의 수학적 값을 비교한다. 비교하는 두 프리픽스의 길이가 다른 경우는 짧은 프리픽스를 기준으로 수학적 값을 비교한다. 만일 비교 부분의 값이 같다면, 길이가 긴 프리픽스의 다음 비트(bit)를 확인하여 그 비트가 0이면 길이가 긴 프리픽스가 작은 프리픽스로 정의되고, 1이면 큰 프리픽스로 정의된다. 또한 프리픽스를 인클로져(enclosure), 인클로즈드(enclosed), 디스조인트(disjoint)로 분류하는데, 인클로져는 자신을 부-스트링(sub-string)으로 갖는 다른 프리픽스가 존재하는 프리픽스이며, 인클로즈드는 인클로져를 부-스트링으로 갖는 프리픽스이고, 디스조인트는 다른 프리픽스와 부-스트링의 관계를 갖지 않는 프리픽스로 정의된다.

이진검색을 수행함에 있어 인클로져 프리픽스는 인클로즈드 프리픽스 보다 먼저 비교되어야 하는데, 검색트리를 구성하는 과정은 다음과 같다. 먼저 디스조인트 프리픽스와 인클로져 프리픽스들을 크기별로 정렬하여 가운데에 위치하는 프리픽스를 선택하여 트리의 노드를 생성한다. 그 프리픽스보다 작은 것은 트리의 왼쪽, 큰 것은 오른쪽에 위치하게 된다. 만일 인클로져가 트리의 노드로 선택이 된 경우에는 그 인클로져를 부-스트링으로 갖는 인클로즈드 프리픽스들을 현재의 리스트에 크기별로 삽입하며, 이 리스트에서 다시 중간 값을 선택하여 트리의 노드를 생성하는 과정을 반복하여 전체 트리를 구성하게 된다. 이 과정을 거쳐 표 1의 프리픽스를 이용하여 이진검색 트리를 구성하면 그림 2와 같다.

이진 검색트리는 이진 트라이와 달리 빈 노드가 존재하지 않아 메모리 요구량이 줄어들며, 트리의 깊이가 이진 트라이보다 감소하기 때문에, 메모리 접근 횟수도 줄어든다. 하지만 인클로져에 속하는 인클로즈드 수의 분포에 따라 이진 트라이와 마찬가지로 트리가 심한 불균형이 될 수 있다.

가중 이진검색 트리[6]에서는 인클로져가 갖는 인클로즈드의 개수까지를 고려하여 인클로즈드를 많이 갖는

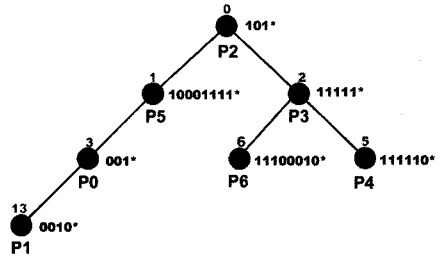

그림 2 이진 검색트리

인클로져를 미리 뽑아 노드를 생성하는 방법에 의하여 기존의 이진 검색 트리에서 발생하는 불균형을 해소하여 보다 균형있는 트리를 구성하였다.

2.1.3 영역분할 이진검색(Binary Search on Range, BSR)

영역분할 이진검색 알고리즘[7]에서는 모든 프리픽스는 $[0, 2^{32}-1]$의 직선상에서 영역으로 표현 될 수 있다는 점을 이용하였다. 프리픽스를 영역으로 표현하기 위하여 프리픽스를 최대 길이로 확장을 하는데, 0으로 채워넣은(padding) 값은 영역의 시작점, 1로 채워넣은 값은 영역의 끝점이 된다. 그림 3에 표 1의 프리픽스를 영역으로 표현하였다.

영역 분할 이진검색 알고리즘에서는 모든 공통 부분을 갖지 않는(disjoint) 영역의 시작점과 끝점이 라우팅 테이블의 엔트리가 되며, 그 엔트리보다 큰 경우와 엔트리의 값과 같은 경우에 대하여 최적으로 일치하는 프리픽스를 미리 계산하여 저장한다. 영역 분할 이진 검색 알고리즘을 사용하여 구성된 라우팅 테이블은 그림 4와 같다.

이 라우팅 테이블에서 이진 검색을 진행하는데, 입력된 주소가 비교되는 엔트리 값보다 커서 검색이 아래로 진행되는 경우에는 엔트리 값보다 클 때 항목의 BMP를 기억한 후 검색을 진행하고 만일 더 이상 검색을 할 엔트리가 존재하지 않을 경우 그 시점에 저장하고 있는 최적으로 일치하는 프리픽스를 최종 검색 결과로 결정하며 검색이 종료된다. 만일 엔트리 값과 입력된 주소가 정확히 일치한다면 엔트리 값과 같을 때 항목의 최적으로 일치하는 프리픽스 값을 최종 검색 결과로 결정하며 검색이 즉시 종료된다.

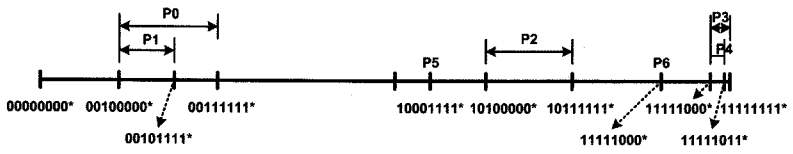

그림 3 영역으로 표현한 프리픽스

값(value)	최적으로 일치하는 프리픽스 (엔트리 값보다 클 때)	최적으로 일치하는 프리픽스 (엔트리 값과 같을 때)
00100000	P1	P1
00101111	P0	P1
00111111	-	P0
10001111	-	P5
10100000	P2	P2
10111111	-	P2
11100010	-	P6
11111000	P4	P4
11111011	P3	P4
11111111	-	P3

그림 4 영역분할 이진 검색 알고리즘의 라우팅 테이블

2.2 해싱구조에 기반한 알고리즘

2.2.1 병렬 해싱(Parallel Hashing) 구조

병렬 해싱 구조[8]는 프리픽스의 길이별로 해싱 하드웨어, 주-테이블, 보조 테이블을 갖는 구조로서, 각 테이블에서 병렬적으로 검색을 수행하여 일치되는 엔트리 중 가장 긴 엔트리를 선택하는 구조이다. 이러한 구조로 인해 최장길이 일치 문제가 확정 일치(extct matching) 문제로의 전환이 가능하게 되었다. 하지만 병렬 검색 구조는 소프트웨어적으로 처리하기가 힘들어 유연성이 떨어지며, 하드웨어의 복잡도가 크다는 단점이 있다. 병렬 해싱의 전체 구조는 그림 5와 같다.

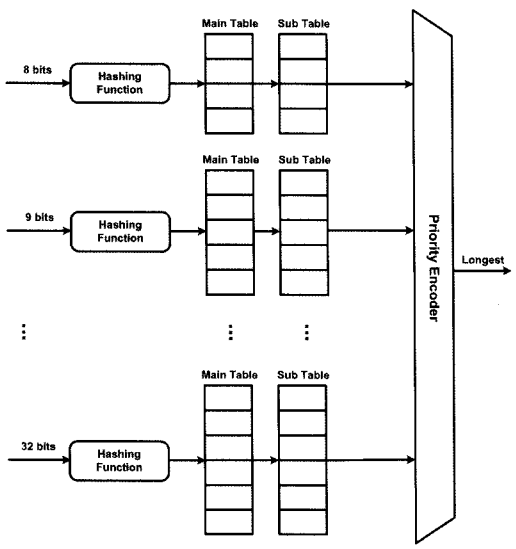

그림 5 병렬 해싱 구조

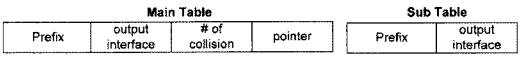

그림 6 병렬 해싱 구조의 메인 테이블과 보조 테이블

이 구조는 각 프리픽스의 길이(8비트~32비트)에 따른 해싱 색인을 구하고 그 색인의 값을 메인 테이블의 포인터로 하여, 그 위치에 프리픽스를 저장한다. 만일 그 위치에 다른 프리픽스가 이미 저장이 되어있다면, 이것은 충돌(collision)이 일어난 경우이며 메인 테이블의 엔트리가 포인터로 가리키고 있는 보조 테이블에 충돌이 난 프리픽스를 크기에 따라 정렬하여 저장한다. 메인 테이블과 보조 테이블의 구조는 그림 6과 같다.

검색과정은 먼저 입력된 주소를 길이별로 구성된 모든 해싱 하드웨어에 입력하여 길이별 해싱 색인을 얻는다. 해싱 색인 값을 포인터로 하여 메인 테이블의 엔트리에 저장된 프리픽스와 입력이 일치하는지 확인한다. 만일 일치하지 않는다면 그 엔트리에 대한 충돌횟수를 확인 후 충돌이 있다면 메인 테이블 엔트리가 가르키는 보조 테이블에서 이진 검색을 수행하여 검색을 진행한다. 길이별 해시 테이블을 별도의 SRAM으로 구성하여 각 길이별로 병렬로 검색을 수행한다. 우선순위 인코더(priority encoder)에서 각 길이별로 일치하는 프리픽스 중 가장 긴 프리픽스를 선택하여 최종 검색 결과로 출력하게 된다.

2.2.2 다중 해싱(Multiple Hashing)

다중 해싱은 완전 해싱 함수를 찾아내는 데 긴 시간이 걸리는 단점을 보완하기 위해, 하나의 완전해싱 함수를 사용하는 대신 여러개의 해싱 함수를 사용하는 것을 제안한 알고리즘[9]이다. 그림 7에 두개의 해싱 함수와 두 개의 테이블을 사용한 경우를 보였다. 하나의 프리픽스에 대하여 두개의 해싱 함수를 거쳐 얻어진 두개의 해싱 색인을 테이블의 포인터로 사용하여, 두 개의 테이블에 접근하고, 해싱 색인 값이 가리키는 테이블의 버킷 중 부하가 적은 테이블에 프리픽스를 저장한다. 이러한 저장 방식으로 인해 충돌 발생이 줄어들고 프리픽스가

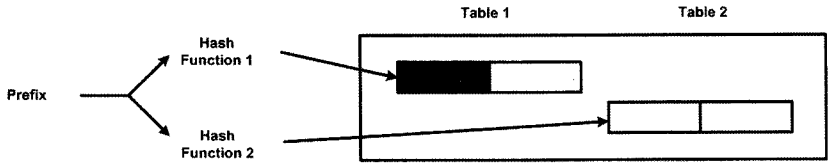

그림 7 다중 해싱 함수를 사용한 테이블 구성

각 테이블에 균일하게 저장되게 된다.

[9]에서는 모든 프리픽스를 16, 24, 혹은 32로 길이를 확장하여 다중 해싱 방법으로 테이블에 저장하고, 프리픽스 길이 레벨에 따른 이진 검색을 통하여 검색을 수행하는 방법을 제안하였다. 이러한 방식의 경우 특정 길이 레벨로 프리픽스를 확장하기 위하여 프리픽스들의 복사가 불가피하다. 또한 충돌이 일어나 프리픽스를 저장하지 못하는 경우에 대한 해결을 제시하지 못하고 있다. 이 경우 프리픽스를 테이블에 모두 저장하기 위해 해싱 함수를 처음부터 다시 찾아야 하며, 이 때문에 새로운 프리픽스를 추가하려 할 때 최악의 경우 모든 테이블을 다시 구성해야 한다는 단점이 있다.

2.2.3 병렬 다중 해싱(Parallel Multiple Hashing)

다중 해싱 구조에 병렬 검색을 접목시킨 구조[10]로서 전체 구조를 그림 8에 보였다. 이 구조에서는 해싱 함수를 위하여 CRC 생성기를 사용하였다. 테이블을 구성하는 방법은 먼저 프리픽스를 CRC 함수에 입력하여 여러 개의 해싱 색인을 뽑는데, 전절에서 언급한 다중 해싱 방법으로 프리픽스를 저장하게 되며, 만일 충돌이 발생하여 주어진 버킷에 프리픽스를 저장하지 못하는 경우에는 오버플로우 테이블에 따로 저장하게 된다. 오버플로우 테이블은 작은 사이즈의 TCAM을 사용하여 구현한다.

검색 진행 과정은 입력된 주소를 CRC 함수에 입력하여 각 길이별 테이블에 해당하는 해싱 색인을 결정한다. 각 길이별로 테이블이 별도의 SRAM에 저장되어 있으므로 여러개의 해싱 색인이 생성됨과 동시에 길이별로 검색이 병렬적으로 진행되며, 오버플로우 테이블도 동시에 검색이 진행된다. 우선 순위 인코더에서 각 테이블에서 결정된 입력 주소와 일치하는 프리픽스 중 가장 길이가 긴 것을 결정하여 최종 검색 결과로 출력한다. 이 구조 또한 병렬 검색에 기초한 구조이므로 소프트웨어적으로 처리하기가 힘들어 유연성이 떨어지며, 하드웨어의 복잡도가 크다는 단점이 있다.

2.3 블룸 필터에 기반한 알고리즘

2.3.1 블룸 필터 이론

블룸 필터는 매우 간단한 비트-벡터(bit-vector)를 사용하여 입력 값이 특정 집합에 속하는 요소(member)인지를 판단하여 주는 필터이다. 블룸 필터의 크기를 M, 블룸 필터에 저장될 요소의 수를 N이라고 할 때, 블룸 필터는 다음과 같이 프로그래밍 한다. 모든 요소에 대하여 k 개의 해싱 함수를 사용하여 k 개의 해싱 색인 값을 구하되 해싱 색인 값은 1부터 M 사이에 있도록 한다. 이 값은 M-bit 블룸 필터의 주소 값으로 사용되며, 이 주소에 해당하는 비트-벡터 값을 1로 셋팅한다. 만일 그 비트-벡터 값이 이미 1로 셋팅되어 있다면, 그 값은 1로 유지된다. 요소 x를 블룸 필터에 프로그래밍하는 과정은 다음과 같다[11].

BFAdd (x)

1) for (i = 1 to k)

2) Vector[h_i (x)] ← 1

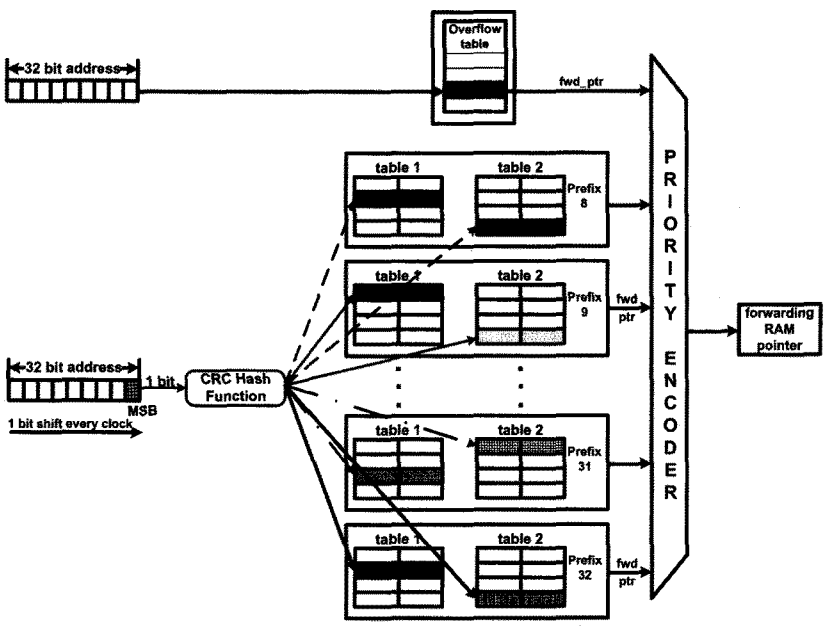

그림 8 병렬 다중 해싱의 구조

어떤 입력 값이 블룸 필터의 요소인지 쿼링(querying)을 통하여 확인할 수 있다. 어떤 입력 값에 대하여 같은 k 개의 해싱 함수를 사용하여 k 개의 해싱 색인 값을 구한다. 이 해싱 색인 값에 해당하는 k 개의 비트-벡터 값 중 하나라도 0이 존재하면 이 입력 값은 블룸 필터의 요소가 아니며, 이러한 경우를 음성(negative)이라고 한다. 모두 1이면 블룸 필터의 요소일 가능성이 있으며 이 경우를 양성(positive)이라고 정의한다. 입력 x 에 대한 쿼링의 과정은 다음과 같다[11].

BFQuery (x)

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

양성의 결과 중 그 입력 값이 블룸 필터의 실제 요소인 것을 참된 양성(true positive)라고 한다. 한 가지 여기서 주의할 점은 쿼링을 할 때, 하나의 입력 값에 대한 k 개의 비트 벡터의 값이 모두 1이어도, 즉 양성 결과이어도 그 입력 값이 블룸 필터의 실제 요소가 아닐 수 있다는 점이다. 이러한 경우를 거짓 양성(false positive)이라고 하는데, 그 이유는 비트-벡터 값이 단 하나의 요소에 의해 프로그래밍 되는 것이 아니라, 다른 요소에 의해 중복 프로그래밍 되었을 수 있기 때문에 이런 거짓 양성을 줄이는 것이 블룸 필터를 구성하는데 중요한 관건이라 하겠다.

2.3.2 블룸 필터 기반의 IP 주소 검색 구조

[11]에서는 블룸 필터를 사용한 IP주소 검색 구조를 제안하였다. [11]의 구조는 그림 9에서 볼 수 있듯이 프리픽스의 길이 별로 블룸 필터를 따로 구성한다. 검색 과정은 각 블룸 필터에서 병렬적으로 쿼링을 하여, 양성으로 확인된 길이들 중 우선 순위 인코더에서 가장 길이가 긴 것부터 차례로 해시 테이블에 접근하여 일치하는 프리픽스의 존재 여부를 확인한다. 이러한 과정을 통하여 검색을 수행하므로 해시 테이블에 접근하여야 하는 프리픽스 길이의 수를 줄일 수 있다는 장점이 있다. 하지만 이 구조에서는 블룸 필터를 프리픽스 길이별로 모두 구성하여야 하므로 구현이 복잡하다. 해시 테이블 또한 길이별로 따로 구성하는 것을 가정하여, 해시 테이블의 수를 줄이기 위하여 통제된 프리픽스 확장(controlled prefix expansion: CPE) 방식을 사용하는 것을 제안하였는데, CPE 방식을 사용할 경우, 프리픽스의 복사가 불가피한 단점이 있다. 또한 [11]의 논문에서는 해시 테이블을 구성함에 있어 충돌(collision) 관리에 대한 제안이 없이 완전해싱 함수를 구하는 것을 가정하였으므로 실용적이지 못하다.

3. 제안하는 구조

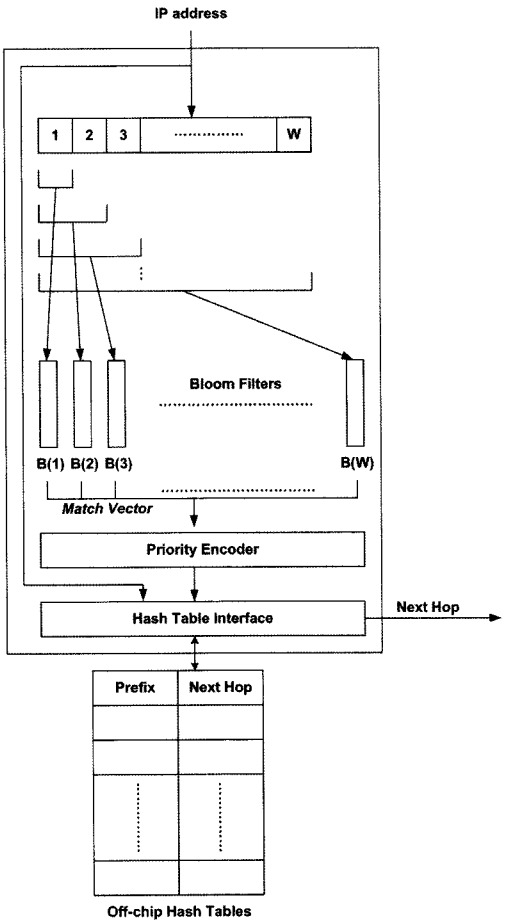

그림 9 블룸 필터 기반의 IP 주소 검색 구조

본 논문에서 제안하는 구조는 통합 블룸 필터와 통합 다중-해시 테이블을 결합하여 효율적인 검색 속도와 간단한 구성을 보이는 IP 주소 검색 구조이다. 제안하는 구조의 장점은 단일 블룸 필터에 다양한 길이의 프리픽스에 관한 정보를 모두 수용하며, 단일 다중-해시 테이블에 다양한 길이의 프리픽스를 모두 저장하여 효율적이며 구현이 매우 간단하다는 점이다. 따라서 소프트웨어, 하드웨어 어느 것으로도 쉽게 구현이 가능하다. 또한 단일 CRC 생성기를 사용하여 그 결과로부터 해싱 색인을 생성 한 뒤, 그 색인을 블룸 필터와 해시 테이블에 모두 적용한다. 그림 10에 본 논문에서 제안하는 IP 주소 검색 구조를 보였다. 제안하는 구조를 하나하나 살펴보면 다음과 같다.

3.1 CRC 해싱 함수

CRC 생성기를 해싱 하드웨어로 사용하면 다양한 길이의 프리픽스에 대해 하나의 기준으로 해싱 색인을 얻어낼 수 있다. 길이가 다른 프리픽스들이 CRC 생성기

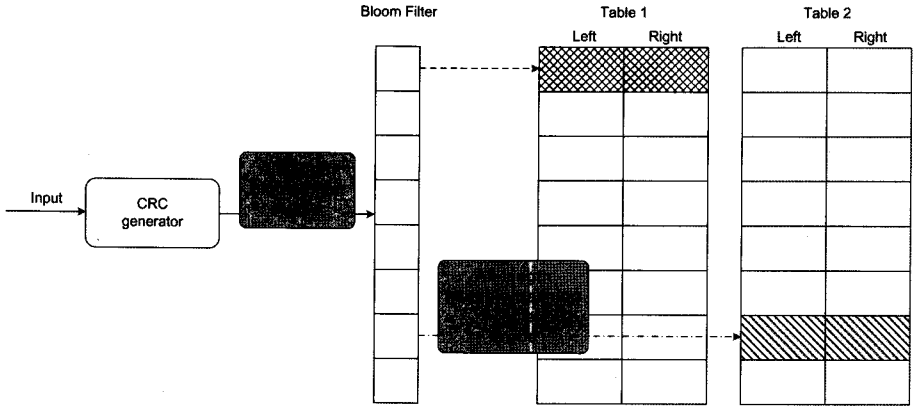

그림 10 제안하는 IP 주소 검색 구조

를 거치면 모두 같은 길이의 스크램블 된 코드가 생성되며, 이를 이용하여 해싱 색인을 구성하게 된다. CRC 생성기에서 생성된 코드는 각 프리픽스를 구성하는 비트를 일정 규칙에 의해 스크램블 한 특정 길이의 데이타이기 때문에, 이 코드값으로부터 어떤 위치에서든 자유롭게 선택하여 원하는 길이의 해싱 색인을 원하는 수만큼 얻어 낼 수 있다는 장점이 있다. 또한 해싱은 프리픽스끼리 충돌이 일어날 수 있는데, 이 충돌을 최소화 할 수 있는 해싱 색인을 구현하는 것이 관건이며, CRC 생성기는 하드웨어로 구현하기 쉬우면서도 충돌이 적은 해싱 색인을 얻어낼 수 있다는 점에서 매우 적합한 구조라고 하겠다.

그림 11은 8비트-CRC 하드웨어 구조를 보이고 있다. CRC내의 모든 플립-플랍(flip-flop)은 0으로 초기화되어 있으며, 입력 값이 한 비트씩 들어오면서 플립-플랍의 값과 XOR 된다. 마지막 비트까지 들어오면 CRC 생성기의 동작이 멈추고 플립-플랍에 저장되어 있는 값이 CRC 코드 값이 된다. 표 1의 프리픽스들에 대하여 그림 11의 8비트-CRC 하드웨어를 거치면 표 2의 코드가 생성된다.

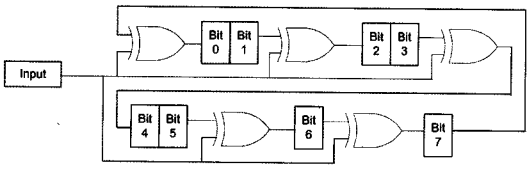

그림 11 8bit-CRC 하드웨어 구조

표 2에서 볼 수 있듯이, 다양한 길이의 프리픽스들이 CRC 생성기를 거쳐 같은 길이의 코드값을 생성하였으며, 이 데이타에서 필요한 비트를 선택하여 여러 개의 해싱 색인을 얻어 낼 수 있다.

표 2 8bit-CRC 코드

Prefix	CRC 코드
001*	10101011
0010*	11010101
101*	01000001
11111*	01011011
111110*	10101101
10001111*	10110110
11100010*	01110001

3.2 통합 블룸 필터

기존에 제안된 블룸 필터는 프리픽스 길이별로 구성되어 메모리를 많이 요구하며 구성에 있어서도 복잡하다는 단점이 있다. 하지만 본 논문에서는 이런 단점을 개선하는 방법으로 하나의 블룸 필터에 다양한 길이의 프리픽스를 프로그래밍하는 방법을 제안한다. 블룸 필터를 구성하기 위해 CRC 생성기를 사용하였으며, 다음과 같은 정의를 사용한다.

1) N: 프리픽스의 개수
2) M: 블룸 필터 사이즈
3) k: 해싱 함수의 개수

N개의 프리픽스를 저장하기 위한 블룸 필터 사이즈(M)는 $2^{\lceil \log_2 N \rceil} = N'$로 하였다. 블룸 필터의 사이즈는 원하는 필터링 성능에 따라 조절 가능하다. 해싱 색인에 필요한 비트 수는 $\lceil \log_2 N \rceil$이 될 것이며, 해싱 함수의 수(k)는 2로 정의하였다. 본 논문에서는 표 3과 같이 해싱 색인에 사용하기 위해 CRC 코드의 첫 $\lceil \log_2 N \rceil$비트와 마지막 $\lceil \log_2 N \rceil$비트를 사용하였다.

블룸 필터를 프로그래밍하는 과정은 다음과 같다. 표 1의 프리픽스의 수(N)는 7이므로 블룸 필터의 사이즈(M=N')는 8이 된다. 블룸 필터의 모든 비트-벡터의 값을 0으로 초기화 한 뒤, 각 색인의 값이 비트-벡터의

표 3 해싱 색인

CRC 코드	Hashing Index 1	Hashing Index 2
10101011	101 (5)	011 (3)
11010101	110 (6)	101 (5)
01000001	010 (2)	001 (1)
01011011	010 (2)	011 (3)
10101101	101 (5)	101 (5)
10110110	101 (5)	110 (6)
01110001	011 (3)	001 (1)

주소이므로 그에 해당하는 비트-벡터의 값을 1로 프로그래밍 한다. 표 1의 모든 프리픽스에 대해 블룸 필터를 프로그래밍 한 결과는 그림 12와 같다.

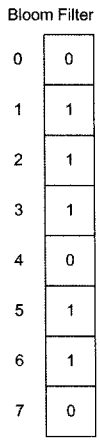

그림 12 프로그래밍 된 블룸 필터

3.3 통합 다중-해시 테이블

해시 테이블도 [8-10]과 같이 길이별로 각각의 테이블을 만드는 알고리즘이 제안되었으나 메모리 요구량이 클 뿐 아니라 여러 개의 SRAM을 사용하여야 하므로 구현이 복잡하다는 단점이 있다. 때문에 [11]에서는 특정 길이의 프리픽스를 기준으로 다른 길이의 프리픽스를 확장하여 해시 테이블을 구성하는 방식인 통제된 프리픽스 확장(controlled prefix expansion: CPE)의 사용이 제안되었다. CPE를 사용하는 경우 프리픽스의 복사가 불가피하여 저장하여야 하는 프리픽스의 개수가 비효율적으로 많아지는 단점이 있다. [12]에서는 프리픽스의 복사를 피하기 위하여 앞의 몇 비트가 같은 프리픽스끼리 묶어 길이의 종류를 줄이고 그 부분을 따로 처리하는 프리픽스 축약(prefix collapsing)을 이용하여 해시 테이블의 수를 줄이는 방법을 제안하였다. 이러한 방식은 해싱의 충돌을 증가시켜 또한 비효율적이다.

본 논문에서는 단일 해시 테이블에 모든 길이의 프리픽스를 저장하여, 메모리 요구량을 줄이며 해시 테이블 구성 또한 용이한 방법을 제안한다. 제안하는 통합 다중-해시 테이블의 구조는 해시 테이블의 개수(해싱 함수 혹은 해싱 색인의 개수), 해시 테이블당 버킷의 개수 및 버킷당 부하의 개수에 따라 요구되는 메모리의 크기가 달라지고, 또한 오버플로우의 개수가 달라진다. 이 절에서는 프리픽스가 N개일 때, 두개의 해싱 함수를 사용하고, 하나의 테이블당 버킷의 수를 $2^{\lceil \log_2 N \rceil}=N'$, 하나의 버킷당 부하의 수를 2로 갖는 해시 테이블을 예로하여 설명한다. 따라서 해싱 색인에 필요한 총 비트 수는 $\lceil \log_2 N \rceil$이 된다. 각 버킷의 두 부하를 그림 10에서 보여주는 바와 같이 각각 Left와 Right로 정의하였다. 여기서 Table1과 Table2는 표 1에서 생성한 두 개의 해싱 색인에 대한 구조이다. 이 두 개의 해시 테이블은 각각 SRAM에 저장 가능하다.

제안하는 해시 테이블에 프리픽스를 저장하는 과정을 보면 다음과 같다. 표 3의 해싱 색인의 값이 해시 테이블의 메모리를 가리키는 포인터가 된다. 해싱 색인 1과 해싱 색인 2를 사용하여 두 개의 테이블에 동시에 접근한다. 접근된 두개의 버킷에 저장된 부하의 갯수를 비교하여 부하의 갯수가 적은 쪽에 프리픽스를 저장한다. 두 개의 버킷의 부하의 수가 같은 경우 임의로 Table1에 저장되는 것을 가정하였다. 만일 두 개의 버킷이 모두 차 있다면 오버플로우 테이블에 프리픽스를 저장한다. 뒤의 성능평가 절에서 보이려는 바와 같이, 통합 다중-해시 테이블을 위해 사용되는 메모리의 크기에 따라 오버플로우의 개수가 조절가능하며, 적절한 크기의 메모리를 사용함에 의하여 오버플로우 된 프리픽스의 개수를 일정 수 이하로 제한할 수 있으므로, 오버플로우 테이블은 캐쉬나 작은 사이즈의 TCAM으로 구현 가능하다.

3.4 검색 과정

본 논문에서 제안하는 방법은 길이별로 구분되지 않은 하나의 블룸 필터와 다중-해시 테이블로 구성되었기 때문에, 최장길이 일치 프리픽스를 찾기 위해서는 입력 값을 통합 블룸 필터와 통합 다중-해시 테이블에 존재하는 프리픽스의 최대 길이부터 순차적으로 감소시키면서 검색을 진행하며, 만일 매치하는 프리픽스가 발견되면 그 프리픽스가 최적으로 일치하는 프리픽스가 되며 그의 포워딩 정보를 출력한다. 여기서 오버플로우 프리픽스를 저장하고 있는 캐쉬나 TCAM은 우선 검색되어 일치하는 프리픽스가 있는지 확인하고, 일치하는 프리픽스가 있다면 그 길이 및 포워딩 정보를 미리 인지하고 있음을 가정한다. 먼저 CRC 생성기에서 구해진 해싱 색인을 사용하여 블룸 필터에서 쿼링을 진행하는데, 입력 값에 대한 두 해싱 색인 값의 위치에 해당하는 비트-벡터의 값 중 0이 존재하면, 이 길이는 일치하는 프리픽스가 없는 길이이므로 해시 테이블에서의 검색을 진

행할 필요가 없다. 비트-벡터의 값이 모두 1인 경우에는 이 길이의 프리픽스가 존재할 가능성이 있으므로 블룸 필터를 통과한다. 이 경우 해시 테이블에서의 검색을 진행하는데 해싱 색인 1의 값으로 Table1의 버킷을 검색하고, 해싱 색인 2의 값으로 Table2의 버킷을 검색하여 입력 값과 일치하는 프리픽스가 저장되어 있는지 확인한다. Table1과 Table2는 별도의 SRAM에 저장되어 있으므로, 검색시 병렬적으로 메모리에 접근하여 검색을 진행하게 된다. 해시 테이블에서의 검색결과 일치하는 프리픽스가 있다면 이는 최장 길이 일치 프리픽스이므로 이 결과를 출력하고 검색을 종료한다. 이 경우 미리 찾아진 오버플로우 테이블의 검색 결과와 길이를 비교하여 더 긴 쪽이 최적으로 일치하는 프리픽스로 선정된다. 해시 테이블에서의 검색결과 일치하는 프리픽스가 없다면 입력 값을 현재 입력된 길이보다 한단계 짧은 프리픽스 길이만큼 잘라 같은 과정을 반복한다.

표 1의 모든 프리픽스를 해시 테이블에 저장한 결과를 그림 13에 보였다. 그림 13을 예로 검색과정을 설명하면 다음과 같다. 입력된 주소가 00100010이라고 가정하면, 통합 블룸 필터와 통합 다중-해시 테이블에 존재하는 프리픽스의 길이가 8, 6, 4, 3이므로, 이 길이만큼 순차적으로 잘라 검색을 진행한다. 따라서 첫번째 검색의 입력 값은 00100010이 되며 이 입력 값의 8비트-CRC 코드는 10001000이다. 이 CRC 코드의 해싱 색인은 4와 0이다. 블룸 필터의 4번째와 0번째의 비트-벡터를 확인하면 0번째 비트-벡터값이 0이므로, 음성으로 판명되고, 이 입력 값은 블룸 필터의 요소가 아니므로 해시 테이블로의 접근이 차단된다. 따라서 입력 값을 한 단계 감소시켜 검색을 다시 진행한다. 입력 값 001000의 8비트-CRC 코드는 01110101이며 해싱 색인은 3과 5이다. 블룸 필터의 3번째 5번째 비트-벡터가 모두 1이므로, 이는 양성으로 판명되고, 입력 값은 블룸 필터를 통과한다. 그러므로 해시 테이블에서 검색을 진행하며, Table1의 3번째와 Table2의 5번째 버킷에 입력 값 001000에 해당하는 프리픽스가 존재하지 않으므로, 이는 거짓 양성이다. 입력 값의 길이를 한 단계 감소시켜 검색을 계속 진행한다. 다음 입력 값은 0010이며 8비트-CRC 코드는 11010101이다. 해싱 색인은 6과 5이며, 양성이므로, 이 값은 블룸 필터를 통과한다. 해시 Table1의 6번째와 Table2의 5번째 버킷을 검색하며, Table1의 6번 버킷에 입력 값과 일치하는 프리픽스 0010*이 존재한다. 그러므로 이는 참된 양성(true positive)이다. 따라서 입력 값 00100010의 LPM이 0010*으로 정해지며 검색이 종료된다. 이 예에서는 모두 세번의 통합 블룸 필터 접근을 통하여, 한번의 음성, 한번의 거짓 양성, 한번의 참된 양성 결과를 얻었으며, 블룸 필터의 양성 결과에 따른 두번의 통합 다중-해시 테이블로의 접근이 수행되었다.

3.5 갱신

제안하는 구조에서 새로운 프리픽스를 추가(announce)하는 과정과 기존의 프리픽스를 제거(withdraw)하는 과정을 살펴보면 다음과 같다. 새로이 추가 할 프리픽스가 CRC 하드웨어를 거치고, 그 아웃풋에서 해싱 색인을 구한다. 해싱 색인으로 통합 블룸 필터를 프로그래밍한다. 다음으로 통합 다중-해시 테이블에서 해싱 색인 값에 위치하는 자리가 비어있다면 해시 테이블에 프리픽스가 추가되며, 만일 해시 테이블의 모든 부하가 차 있다면 오버플로우 테이블에 저장된다. 프리픽스 제거의 경우 CRC 아웃풋에서 해싱 색인을 구한 뒤, 그 값으로 통합 다중-해시 테이블에서 검색을 진행하며, 4개의 부하중 제거 할 프리픽스가 발견되면 그 프리픽스를 제거한다. 이 경우 추가의 경우와 달리 블룸 필터는 프로그래밍하지 않는다. 따라서 통합 블룸 필터의 거짓 양성이 증가할 수 있지만, 통합 다중-해시 테이블에서의 최적으

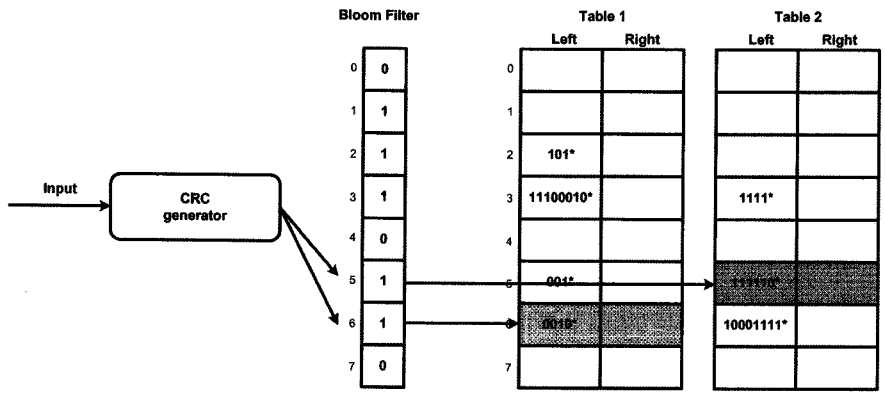

그림 13 제안하는 구조의 검색 진행 과정

로 일치하는 프리픽스 검색의 정확성에는 영향을 미치지 않는다.

4. 제안하는 구조의 성능 평가

본 논문에서는 실제 백 본 라우터에서 사용된 다양한 크기의 라우팅 테이블[13]을 가지고 성능을 실험하였다. 앞의 2.1절과 2.2절에서 소개한 알고리즘들과 성능을 비교하였으며, 메모리 접근 회수를 기준으로 비교하였다. 메모리를 읽고 쓰기 위하여 메모리에 접근하는 회수는 관련 연구들에서 검색 성능의 비교를 위하여 일반적으로 사용되어온 방법으로서, 이는 검색과정에 있어 메모리 접근에 소요되는 시간이 가장 크며, 메모리 접근 회수가 많을수록 입력에 대해 일치하는 값을 찾기까지 소요되는 시간이 증가하기 때문이다. 실제 라우팅 테이블에 존재하는 프리픽스들은 최대 길이가 32비트이므로 시뮬레이션에서는 32비트-CRC를 사용하였다.

본 논문에서는 먼저 기존의 길이별 블룸 필터[11]와 제안하는 통합 블룸 필터의 검색 성능을 비교하였다. 우선 공평한 비교를 위해 실제 백 본 라우터에서 사용하는 라우팅 테이블 중 227,223개의 프리픽스를 갖는 Telstra 데이타를 이용하였으며, 두 구조의 메모리 요구량이 같도록 하기 위해 라우팅 데이타 중 일부를 선택하였다. 다양한 프리픽스 길이 중 13개의 길이를 선택하여, 각 길이에 해당하는 프리픽스의 개수가 2의 승수가 되도록, 또한 총 프리픽스의 개수도 2의 승수가 되도록 프리픽스를 선택하였다. 본 논문에서는 총 2^{17}개의 프리픽스를 사용하였다. 길이별 블룸 필터의 경우 13개의 블룸 필터에, 제안하는 블룸 필터의 경우 한 개의 블룸 필터에 2^{17}개의 프리픽스를 저장하였다.

두 구조의 성능 비교 결과는 표 4와 같다. 블룸 필터 사이즈(m)를 n, $2n$, $4n$, $8n$, $16n$으로 증가시키면서 성능을 비교하였다. 여기서 모든 입력 값의 참된 양성에 대한 메모리 접근 횟수는 한번이다. 다시 말하면 모든 입력 값에 대하여 참된 양성이 나오면, 그 입력 값에 대한 검색이 종료된다. 실험 결과 두 구조의 성능에 큰 차이를 보이지 않았으며, 블룸 필터의 사이즈가 n, $2n$, $4n$, 일 때에는 기존의 구조보다 본 논문에서 제안하는 구조의 양성 비율이 낮아 필터링 능력이 높았으며, 블룸 필터의 사이즈가 $8n$, $16n$일 때 기존의 길이별 블룸 필터의 필터링 능력이 조금 더 높아짐을 알 수 있다. 하지만 기존의 구조는 구현하는 데 있어, 제안하는 구조에 비해 훨씬 복잡하다는 점을 감안했을 때 제안하는 구조가 구현도 쉬우며, 필터링 능력도 우수한 구조라는 것을 알 수 있다.

다음으로 본 논문에서 제안하는 통합 블룸 필터와 통합 다중 해싱 구조에 대해 실제 백 본 라우터에서 사용되는 데이타를 사용하여 실험하였다.

표 5는 실험에 사용한 라우팅 테이블들의 오버플로우 개수를 나타내고 있다. 테이블 사이즈(Table Size)는 통합 다중-해시 테이블의 엔트리 개수를 나타낸다. 프리픽스 개수 N에 대하여 통합 다중-해시 테이블의 엔트리 개수를 $2^{\lfloor \log_2 N \rfloor}$으로 했을 때와 $2^{\lceil \log_2 N \rceil}$으로 했을 때의 오버플로우 개수 및 메모리 사용량을 비교하여 보여주고 있다. 제안하는 구조에서의 메모리 사용량은 오버플로우 개수와 트레이드 오프(trade-off) 관계에 있으며 적절한 크기의 메모리를 사용함에 의하여 오버플로우의 개수를 조절할 수 있음을 알 수 있다. 예를 들어 테이블의 엔트리 개수를 $2^{\lceil \log_2 N \rceil}$을 사용한 경우에는 PORT80에서 발생한 단 하나의 오버플로우를 제외하고는 전혀 오버플로우가 발생하지 않았음을 알 수 있다.

제안하는 구조의 통합 다중-해시 테이블의 엔트리는 그림 14와 같은 구조를 갖는다.

표 6과 그림 15에 제안하는 통합 블룸 필터 구조의 성능을 보였다. 통합 다중-해시 테이블의 엔트리 개수를

표 4 길이별 블룸 필터와 본 논문에서 제안하는 통합 블룸 필터의 성능 비교

Routing Table	no. of prefix (n)	m	no. of bits (S)	Mem (kbyte)	no. of input probe		no. of positive			
							PT_{max}		PT_{avg}	
					IT_{max}	IT_{avg}	[11]	Prop.	[11]	Prop.
a subset of Telstra	131,072	n	2^{17}	16	13	6.30	13	13	5.15	4.93
		$2n$	2^{18}	32			13	13	3.24	3.04
		$4n$	2^{19}	64			10	9	1.85	1.74
		$8n$	2^{20}	128			7	9	1.29	1.39
		$16n$	2^{21}	256			4	8	0.98	1.25

표 5 제안하는 해시 테이블의 엔트리 수와 오버플로우 수, 메모리 사용량

	N	N'	Table Size	No. of overflow	메모리 사용량 (Mbyte)
MAE-WEST1	14553	8192	2^{13}	35	0.18MB + 35 entry TCAM
		16384	2^{14}	0	0.36MB
Aads	20204	16384	2^{14}	1	0.36MB + 1entry TCAM
		32768	2^{15}	0	0.72MB
MAE-WEST2	29584	16384	2^{14}	0	0.36MB
		32768	2^{15}	0	0.72MB
MAE-EAST1	37993	32768	2^{15}	1	0.72MB + 1 entry TCAM
		65536	2^{16}	0	1.44MB
MAE-EAST2	39464	32768	2^{15}	2	0.72MB + 2 entry TCAM
		65536	2^{16}	0	1.44MB
PORT80	112310	65536	2^{16}	176	1.44MB + 176 entry TCAM
		131072	2^{17}	1	2.88MB + 1entry TCAM
Grouptlcom	170601	131072	2^{17}	32	2.88MB + 32 entry TCAM
		262144	2^{18}	0	5.75MB
Telstra	227223	131072	2^{17}	48	2.88MB + 48 entry TCAM
		262144	2^{18}	0	5.75MB

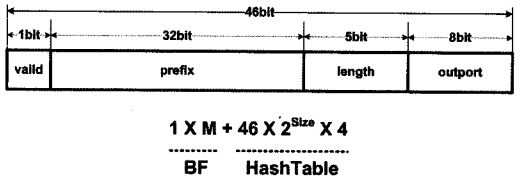

그림 14 통합 다중-해시 테이블의 엔트리 구조

$N'=2^{\lceil \log_2 N \rceil}$으로 한 경우에 대하여 성능을 정리하였으며, 블룸 필터(M)의 크기는 N', $2N'$, $4N'$, $8N'$, $16N'$으로 증가시켜 가며 시뮬레이션을 수행하였다. 여기서 해시 테이블 접근 비율(%)은 블룸 필터에서의 쿼링을 할 때, 블룸 필터 접근(probe) 대비 다중-해시 테이블 접근 비율이며, M이 N'에서 $16N'$으로 증가하면서 비율이 급격히 감소하는 것을 볼 수 있다. 블룸 필터의 크기가 증가함에 따라 거짓 양성이 현저히 줄어들고, 따라서 블룸 필터의 필터링 능력이 증가하여 해시 테이블로의 불필요한 메모리 접근을 막는 것을 알 수 있다. 이는 블룸 필터에서 음성의 비율이 증가하고, 거짓 양성이 감소하는 것을 의미하며 블룸 필터의 사이즈가 커질수록 블룸 필터의 필터링 능력이 증가한다는 것을 의미한다. 블룸 필터의 크기가 $16N'$인 경우에 평균 1.04-1.17번의 다중-해시 테이블 접근에 의하여 IP 주소 검색이 가능함을 볼 수 있다.

그림 16, 그림 17에서 통합 블룸 필터의 사이즈가 증가할 때, 제안하는 통합 다중-해시 테이블의 평균 접근 회수(HT_{avg})와 메모리 요구량을 비교하였다. 블룸 필터의 크기가 커짐에 따라 다중-해시 테이블로의 메모리 접근 회수가 작아지는 것을 볼 수 있다. 그림 17의 메모리 요구량으로부터 블룸 필터의 크기는 전체 메모리 요구량에 거의 영향을 미치지 않음을 보여주고 있다.

앞의 2.1절에서 소개한 알고리즘들과 제안하는 알고리즘의 성능을 비교하였다. 여기서 성능비교에 사용된 제안하는 구조는 블룸 필터 크기는 M=16N'이고, 통합 다중-해시 테이블의 크기는 N'인 경우로서, PORT80 라우팅 테이블에서 발생한 한 개의 오버플로우를 제외하

표 6 제안하는 통합 블룸 필터와 통합 다중-해시 테이블 구조의 성능

		통합 블룸 필터 접근 수			통합 해시 테이블 접근 수			Hash table access rate(%)
Routing Table (N)	N'	M	BT_{max}	BT_{avg}	Size	HT_{max}	HT_{avg}	
MAE-WEST1 (14553)	16384	N'	22	8.16	2^{14}	19	5.97	73.2
		$2N'$				15	3.49	42.8
		$4N'$				12	1.98	24.3
		$8N'$				9	1.29	15.8
		$16N'$				6	1.09	13.4
Aads (20204)	32768	N'	20	5.88	2^{15}	14	3.49	59.4
		$2N'$				11	2.06	35.0
		$4N'$				7	1.35	23.0
		$8N'$				4	1.11	18.9
		$16N'$				4	1.04	17.7
MAE-WEST2 (29584)	32768	N'	24	9.98	2^{15}	22	7.33	73.4
		$2N'$				18	4.22	42.3
		$4N'$				9	2.19	21.9
		$8N'$				7	1.37	13.7
		$16N'$				6	1.11	11.1
MAE-EAST1 (37993)	65536	N'	22	7.99	2^{16}	19	4.39	54.9
		$2N'$				15	2.38	29.8
		$4N'$				9	1.47	18.4
		$8N'$				7	1.15	14.4
		$16N'$				6	1.06	13.3
MAE-EAST2 (39464)	65536	N'	22	7.88	2^{16}	18	4.45	56.5
		$2N'$				14	2.43	30.8
		$4N'$				8	1.49	18.9
		$8N'$				8	1.16	14.7
		$16N'$				7	1.06	13.5
PORT80 (112310)	131072	N'	25	11.96	2^{17}	24	8.35	69.8
		$2N'$				20	4.69	39.2
		$4N'$				20	2.40	20.1
		$8N'$				16	1.48	12.4
		$16N'$				15	1.17	9.8
Grouptlcom (170601)	262144	N'	20	6.77	2^{18}	18	4.07	60.1
		$2N'$				16	2.33	34.4
		$4N'$				15	1.47	21.7
		$8N'$				15	1.16	17.1
		$16N'$				13	1.07	15.8
Telstra (227223)	262144	N'	25	9.43	2^{18}	24	6.75	71.6
		$2N'$				21	3.91	41.5
		$4N'$				17	2.11	22.4
		$8N'$				18	1.38	14.6
		$16N'$				16	1.14	12.1

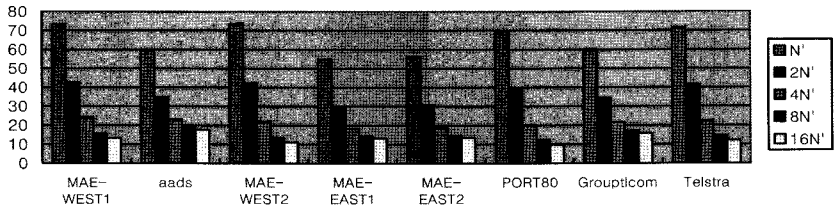

그림 15 제안하는 구조의 해시 테이블 접근 비율(%) 비교

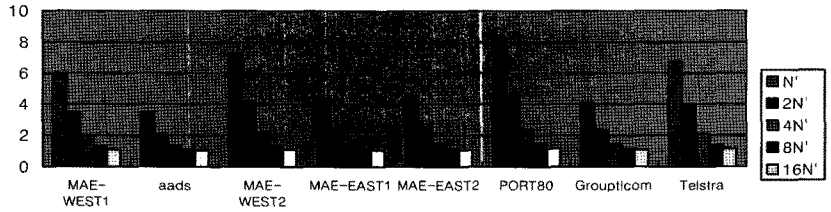

그림 16 블룸 필터의 크기에 따른 다중-해시 테이블의 평균 접근회수(HT_{avg}) 비교

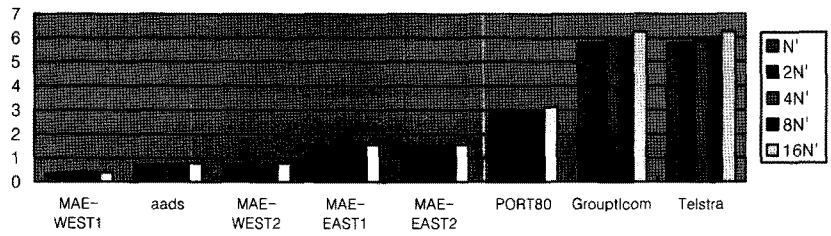

그림 17 블룸 필터의 크기에 따른 메모리 요구량 비교(Mbyte)

고는 오버플로우가 전혀 발생하지 않은 경우이다. 그림 18은 IP 주소 검색을 위해 소요되는 평균 메모리 접근 회수를 비교하였는데, 제안하는 구조의 경우 블룸 필터는 매우 적은 메모리를 소요하여 캐쉬에 저장이 가능하므로, 메모리 엑세스 회수에 포함시키지 않았으며, 해시 테이블의 평균 접근 횟수값으로 비교하였다. 2.1절에서 언급한 트리 구조도 상위 노드들을 캐쉬에 저장하여 검색 성능을 높일 수 있으나, 본 논문에서 제안하는 구조와 같이 통합 블룸 필터와 통합 다중-해시 테이블이라는 명백히 분리 가능한 구조가 아니므로, 본 논문의 검색 성능 비교는 기본 트리 구조들과 비교하였다. 그림 18에서 보여주듯이 제안하는 구조는 타 구조에 비하여 12-27배 적은 수의 메모리 접근에 의하여 IP 주소 검색이 가능함을 알 수 있다.

그림 19는 프리픽스 하나당 요구되는 메모리량를 비교하였는데, 제안하는 구조는 타 구조에 비하여 2-3 배 정도의 메모리를 요구함을 알 수 있다. 앞서 언급한 바와 같이 메모리 사용량은 오버플로우 개수와 트레이드 오프의 관계를 갖는다. 그림 19에서 보여준 실험 결과는 오버플로우의 개수를 최소화하여 PORT80의 경우에 발생한 한 개의 오버플로우를 제외하고는 오버플로우가 발생하지 않은 경우이다. 그림 19에서 보여준 바와 같이

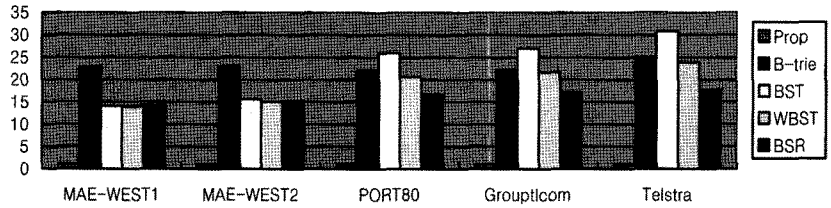

그림 18 기존의 이진 검색 알고리즘과 평균 메모리 접근 횟수

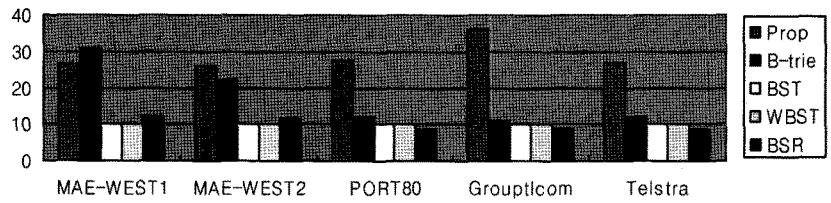

그림 19 기존의 이진 검색 알고리즘과 프리픽스 당 메모리 요구량(byte)

메모리 사용량을 반으로 줄일 수 있으며, 이 경우 타 구조와 비슷한 정도의 메모리를 요구하며, 이 때 최대 176개까지의 오버플로우 프리픽스를 관리하여야 한다.

다음으로 2.2절에서 언급한 해시 기반 알고리즘과 제안하는 구조의 성능을 비교하였다. 2.2절에서 언급한 병렬 검색 구조는 하드웨어의 복잡도가 크며 소프트웨어적으로 처리하기 힘들어 유연성이 떨어진다는 점과 본 논문에서 제안하는 구조는 소프트웨어, 하드웨어 어느것으로도 구현이 쉽다는 것을 고려하여 제안하는 구조의 검색 방법과 동일한 조건으로 순차 검색을 이용하여 성능을 비교하였다.

먼저 표 7에서는 통합 다중-해시 테이블의 엔트리 개수를 $N'=2^{\lceil \log_2 N \rceil}$로 하였을 때의 오버플로우 수와 [8]과 [10]의 오버플로우 수를 비교하였다. [8]의 경우 프리픽스들이 다른 알고리즘에 비해 서브 테이블에 많이 저장된다는 것을 알 수 있으며, 이것이 메모리 접근 횟수를 증가시키는 원인이 된다.

다음으로 메모리 접근 횟수에 대한 비교 결과를 그림 20에 보였다. 제안하는 구조는 타 구조에 비해 6-12배 적은 수의 메모리 접근에 의하여 IP 주소 검색이 가능함을 알 수 있다.

표 7 제안하는 구조와 해시 기반 알고리즘의 오버플로우 수 비교

	# prefix	prop	parallel hashing[8]	parallel multiple hashing[10]
MAE-WEST1	14553	0	3357	0
MAE-WEST2	29584	0	8976	0
PORT80	112310	1	31187	0
Grouptlcom	170601	0	44862	0
Telstra	227223	0	59449	0

그림 21은 프리픽스 하나당 요구되는 메모리량를 비교하였는데, 제안하는 구조는 [8]에 비해 요구되는 메모리 양이 많았으나, [10]에 비해서는 1.5배까지 적게 요구하는 것을 볼 수 있다.

5. 결 론

IP 주소 검색에서 제안되어 온 알고리즘 중 해싱을 이용한 알고리즘은 메모리 접근 횟수가 작아 라우터의 포워딩 속도를 빠르게 해주는 알고리즘으로 알려져있으나, 프리픽스 길이별로 해시 테이블을 구성해야 한다는 어려움이 있다. 또한 기존의 블룸 필터는 메모리 요구량이 매우 작지만 역시 프리픽스 길이별로 구성된 구조여서 구현에 어려움이 있었다. 본 논문에서는 해싱기반의 타 알고리즘에 비해 소프트웨어, 하드웨어 어느 것으로도 구현이 쉬운 용이성을 보이며, 라우터에 적용하였을 때 포워딩 속도를 향상시킬 알고리즘을 제안하였다. 제안하는 구조는 다양한 길이의 프리픽스들을 단일 블룸 필터에 프로그래밍 하고, 다양한 길이의 프리픽스들을 단일 다중-해시 테이블에 저장하는 방법으로서, 실제 백본 라우터 데이타로 주소 검색을 수행하여 실험해 본 결과 매우 빠른 검색 성능을 보임을 확인하였다.

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.

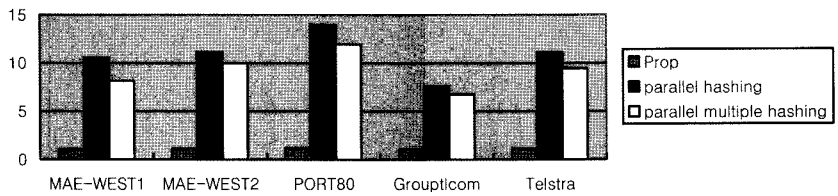

그림 20 제안하는 구조와 해시 기반 알고리즘의 평균 메모리 접근 횟수

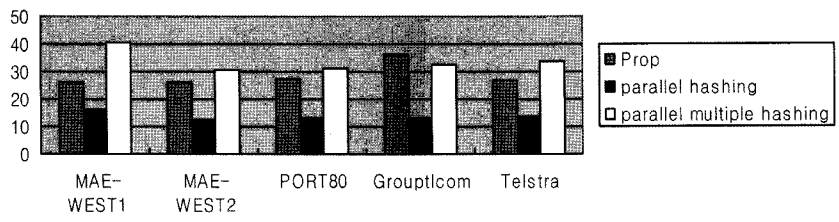

그림 21 제안하는 구조와 해시 기반 알고리즘의 프리픽스 당 메모리 요구량(byte)

[2] H. J. Chao, "Next generation routers," Proc. of the IEEE, Vol.90, No.9, pp. 1518-1558, Sep. 2002.
[3] S. Nilsson and G. Karlsson, "IP address lookup using LC-tries," IEEE Journal on Selected Area in Communication, Vol.17, pp. 1083-1092, June. 1999.
[4] Hyesook Lim and JuHyoung Mun, "An efficient IP address lookup algorithm using a priority trie," Proc. GLOBECOM 2006. pp.1-5.
[5] N. Yazdani and P. S. Min, "Fast and scalable schemes for the IP address lookup problem," Proc. IEEE HPSR2000, pp. 83-92.
[6] Changhoon Yim, Bomi Lee, and Hyesook Lim, "Efficient binary search for IP address lookup," IEEE Communications Letters, Vol.9, No.7, pp. 652-654, Jul. 2005.
[7] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," IEEE/ACM Trans. Networking, Vol.7, No.3, pp. 324-334, Jun. 1999.
[8] 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.
[9] Andrei Broder and Michael Mitzenmacher, "Using multiple hash functions to improve IP lookups," Proc. IEEE Infocom 2001, Vol.3, pp. 1454-1463.
[10] Hyesook Lim and Yeojin Jung, "A parallel multiple hashing architecture for IP address lookup," Proc. HPSR 2004, pp. 91-95.
[11] 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.
[12] 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.
[13] http://www.potaroo.net

박 경 혜
2007년 2월 이화여자대학교 정보통신학과, 학사. 2007년 3월~현재 이화여자대학 교 전자정보통신학과 석사과정. 관심분야는 Router나 switch 등의 Network 관련 SoC 설계, TCP/IP 관련 하드웨어 설계

임 혜 숙
1986년 2월 서울대학교 제어계 측공학과, 학사. 1986년 8월~1989년 2월 삼성휴렛팩커드, 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. 관심분야는 Rourer나 switch 등의 Network 관련 SoC 설계, TCP/IP 관련 하드웨어 설계