고속 라우터의 기가비트 포워딩 검색을 위한 비트-맵 트라이 구조

(The Bit-Map Trie Structure for Giga-Bit Forwarding Lookup in High-Speed Routers)

오 승 현 † 안 종 석 ††
(Seung-Hyun Oh)(Jong-Suk Ahn)

요 약 최근 들어 특별한 하드웨어나 새 프로토콜의 도움 없이 고속 라우터의 포워딩 검색을 지원하는 포워딩 테이블에 대한 연구가 다양하게 진행되고 있다. 본 논문에서는 소프트웨어를 기반으로 일반적인 펜티엄 프로세서에서 기가비트급 포워딩 검색을 지원할 수 있는 새 포워딩 테이블 자료구조를 제시한다. 포워딩 검색은 테이블의 크기에 비례해서 복잡도가 증가하는 라우터 성능의 병목지점으로 알려져 있다. 기존의 소프트웨어를 기반으로 하는 포워딩 검색 연구들은 포워딩 테이블 자료구조로 패트리샤(Patricia) 트라이와 그 변형을 이용하거나, 프리픽스(Prefix) 길이를 키로 해서 함수를 구성하는 방법 등을 사용하여왔다. 본 논문에서 제안된 포워딩 테이블 자료구조는 라우팅 테이블의 프리픽스를 완전이진 트라이로 구성한 후 트라이의 구조와 각 노드별로 링크 되어있는 라우팅 테이블 포인터 정보를 비트열로 표현하여 포워딩 테이블을 구성한다. 트라이의 구조와 라우팅 프리픽스 포인터 정보는 배열이나 링크드-리스트(Linked-list)로 표현하면 대량의 저장공간을 필요로 하지만 제안된 자료구조에서는 각 정보가 하나의 비트로 표현되므로 작은 저장공간으로 충분하며, 또한 트라이를 중간 레벨에서부터 검색할 수 있는 방법을 이용하여 트라이 검색경로를 단축할 수 있다. 비트-맵 트라이로 명명된 이 방법은 백본 라우터의 대용량 라우팅 테이블을 펜티엄 프로세서의 L2 캐쉬에 저장할 수 있는 작은 크기로 압축하고, 검색경로를 단축함으로써 일반적인 펜티엄 프로세서를 이용하여 고속의 포워딩 엔진을 구현할 수 있음을 보여준다. 제안된 방법의 성능을 평가하기 위해서 실제 라우팅 테이블을 대상으로 실험한 결과 초당 5.7백만 번의 라우팅 검색성능을 기록하였다.

Abstract Recently much research for developing forwarding table that support fast router without employing both special hardware and new protocols. This article introduces a new forwarding data structure based on the software to enable forwarding lookup to be performed at giga-bit speed. The forwarding table is known as a bottleneck of the routers performance due to its high complexity proportional to the forwarding table size. The recent research that based on the software uses a Patricia trie and its variants, and also uses a hash function with prefix length key and others. The proposed forwarding table structure construct a forwarding table by the bit stream array in which it constructs trie from routing table prefix entries and it represents each pointer pointing the child node and the associated forwarding table entry with one bit. The trie structure and routing prefix pointer need a large memory when representing those by linked-list or array, but in the proposed data structure, the needed memory size is small enough since it represents information with one bit. Additionally, by use a lookup method that start searching at desired middle level we can shorten the search path. The introduced data structure, called bit-map trie shows that we can implement a fast forwarding engine on the conventional Pentium processor by reducing the backbone routing table fits into Level 2 cache of Pentium II processor and shortens the searching path. Our experiments to evaluate the performance of proposed method show that this bit-map trie accomplishes 5.7 million lookups per second.

• 본 논문은 '99년도 정보통신부 "대학기초연구지원사업"의 일부 지원을 받았음.

† 학생회원 : 동국대학교 컴퓨터공학과
shoh@dongguk.edu

†† 종신회원 : 동국대학교 컴퓨터공학과 교수
jahn@dongguk.edu

논문접수 : 2000년 11월 14일
심사완료 : 2001년 3월 14일

1. 서 론

최근 들어 인터넷 백본이 기가비트급 링크 위주로 구성되고 인터넷 트래픽이 폭발적으로 증가함에 따라 기가비트급 고속 라우터를 개발하기 위한 연구[1]가 활발하게 진행되고 있다. 또한 실시간 전송을 요구하는 멀티미디어 응용서비스가 폭넓게 확산되고, 초고속 가입자망(Access network)이 빠르게 보급되면서 폭증하는 트래픽은 인터넷 백본용량을 기가비트급으로 확장하도록 요구하고 있다. 이러한 현실에 부응하여 미국에서 추진중인 실험적 차세대 인터넷인 인터넷2[2]의 경우에는 622Mbps 링크로 백본을 구성하고 있다. 더구나 웹 응용 프로그램의 영향으로 트래픽이 근거리 망 내부에서 처리되는 경우 보다 광대역 망 백본을 경유하는 비율이 계속적으로 증가하고 있다.

이와 같은 인터넷 백본의 고속화는 백본 라우터에서의 병목현상 발생을 해소하기 위해 기가비트급 입력링크 속도의 총량을 처리할 수 있는 고속 라우터 개발에 강력한 동기를 제공하고있다. 기가비트급 고속 라우터 연구는 하드웨어적으로 처리 가능한 스위칭 조직(Switching fabric) 보다는 라우터 성능의 병목지점인 포워딩 검색 과정의 고속화에 초점을 맞추고 있다. 포워딩 검색은 수신된 IP 패킷의 목적지 IP 주소와 가장 긴 부분이 일치하는 라우팅 테이블 엔트리를 검색하여야 하는 최장 프리픽스 검색(LPM: Longest prefix matching)으로 일정한 길이의 엔트리를 검색하는 전통적인 일치검색(Exact matching) 문제와는 달리 O(1)의 성능을 얻기가 매우 어렵다. 더구나 포워딩 검색의 복잡도는 테이블의 크기와 검색 대상인 IP 주소의 길이에 비례해서 증가한다. 참고로 중대형 이상의 라우터 박스에서는 라우팅 프로토콜의 라우팅 정보 수집 및 교환의 결과로 생성·유지되는 라우팅 테이블과 포워딩 엔진(IP 패킷의 목적지 주소에 따라 출력 링크를 결정)에서 사용하는 포워딩 테이블이 별도로 존재하는 것이 일반적이다. 이때 포워딩 테이블은 라우팅 테이블의 내용이 포워딩 엔진에서 사용하기 적합한 형태로 재구성된 것으로 생각할 수 있다. 본 논문은 포워딩 테이블의 구성방법과 검색에 대한 연구로써 백본 포워딩 테이블을 펜티엄 프로세서[3]의 L2 캐쉬에 저장할 수 있는 작은 크기로 압축함으로써 기가비트 속도로 포워딩 검색을 수행할 수 있는 새로운 포워딩 테이블 자료구조를 제시하고자 한다.

기존의 기가비트급 포워딩 연구는 빠른 접근속도를 갖는 SRAM, CAM과 같은 하드웨어를 사용하는 연구들[4,5,6,7,8,9]과 새 프로토콜을 고안하여 사용하는 연구[10] 및 포워딩 테이블의 자료구조를 개선하여 고속의 검색속도를 얻는 소프트웨어 중심의 연구[11,12,13,14,15,16,17,18]들이 있다. 또한 MPLS[19]와 같이 3계층에서 발생하는 포워딩 검색을 회피하는 방법[20,21]들도 프로토콜 중심의 연구범위에 포함할 수 있다. 본 논문에서 제안하는 새 포워딩 테이블 구조의 이름은 비트-맵 트라이(Bit-map trie)로, 라우팅 테이블의 모든 프리픽스 엔트리를 트라이 구조를 기반으로 하는 비트 마스크 배열로 변환하여 포워딩 테이블로 사용한다.

트라이의 각 노드는 라우팅 프리픽스의 한 비트와 대응되며, 리프노드와 특정한 중간 노드들은 라우팅 프리픽스와 링크되어 라우팅 프리픽스 포인터 정보를 제공한다. 기존의 연구들에서 트라이를 포워딩 테이블의 자료구조로 사용하기 위해서는 트라이의 구조에 대한 정보와 라우팅 프리픽스 포인터 정보를 링크드-리스트 또는 배열로 변경하여 사용하는데, 이때 매우 큰 저장공간을 필요로 하며 많은 비교 연산이 발생하는 단점이 있다. 비트-맵 트라이는 저장공간의 축소를 위해 리프노드와 특정 중간 노드에 대응하는 라우팅 테이블 포인터 정보와 트라이 구조에 대한 링크정보를 비트열로 저장한다. 비트열은 트라이 레벨이 8의 배수가 되는 곳마다에서 수집되므로 256비트가 하나의 비트열이 되어 포워딩 검색 시 IP주소의 8비트 단위가 키로 사용된다.

또한 비트-맵 트라이는 프리픽스 검색을 위해 최상위 노드부터 한 비트씩 탑-다운 방향으로 검색하여야 하는 전통적인 트라이와는 달리 임의의 중간레벨 노드에서 검색을 시작할 수 있다. 즉, 루트노드에서 리프노드까지 한 노드(비트)씩 비교·검색하는 경우 최대 1.44logN (N은 엔트리 개수)[16]의 비교연산이 발생하는데, 이 비교연산 회수를 줄일 수 있도록 트라이의 임의의 중간 레벨로부터 검색을 시작하여 검색속도를 빠르게 할 수 있다. 그 결과 비트-맵 트라이는 포워딩 테이블을 펜티엄 프로세서의 L2 캐쉬에 저장할 수 있는 작은 크기로 압축할 수 있으며, IPv6과 같이 많은 단계의 노드를 갖는 트라이에서도 신속하게 검색범위를 축소하여 빠른 포워딩 검색을 할 수 있다. 본 논문의 포워딩 테이블의 실험에서는 실제 수집된 IP주소를 사용하여 라우팅 테이블의 검색 성공률을 실제 상태와 같이 100%에 근접한 상태를 만들어서 제안된 자료구조의 성능을 측정하였다. 이 것은 기존연구들이 모두 랜덤함수에 의해 생성된 IP주소를 사용하여 얻어진 결과임을 감안할 때 보다 현실에 근접한 결과를 보여주는 중요한 기여라고 판단된다.

본 논문의 구조는 다음과 같다. 2장에서는 관련연구를 살펴보고, 3장에서는 본 논문에서 제안하는 새로운 포워딩 테이블 자료구조인 비트-맵 트라이와 검색 알고리즘에 대해 설명한다. 4장에서는 비트-맵 트라이의 포워딩 검색 실험 결과를 제시하고, 5장에서는 비트-맵 트라이 확장성을 살펴본다. 마지막으로 6장에서는 결론과 향후 연구과제를 제시한다.

2. 관련연구

포워딩 검색에 관련된 기존의 연구는 크게 세 가지로 구분할 수 있다. 첫째 하드웨어 기반 연구, 둘째 소프트웨어 기반 연구, 마지막으로 프로토콜 기반 연구가 있다. 세 가지 부류의 기존연구에 대해 간단히 소개한다.

2.1 하드웨어 기반 연구

하드웨어를 이용한 고속 포워딩 검색 연구는 고속의 SRAM을 이용하는 방법[1]으로 10ns 접근시간을 가진 SRAM에 전체 라우팅 테이블을 저장할 경우 200ns 속도로 IP주소 검색을 할 수 있다. 그러나 SRAM은 고가의 하드웨어라는 단점 때문에 소규모 기업급 라우터 이외에는 적용이 어렵다. [7]에서는 CAM(Content-Addressable Memory)을 이용하고 있는데 CAM은 프리픽스를 길이별로 각각의 CAM에 저장한 상태로 검색할 IP주소와 병렬로 비교 검색한다. 따라서 32비트 주소 검색을 위해서는 32개의 CAM이, 128 비트 주소를 검색하기 위해서는 128개의 CAM이 필요하게 되므로 매우 비싼 시스템이 된다. 보다 저렴한 CAM 구성방법으로 즉, 하나의 CAM에 모든 길이의 프리픽스를 저장하는 기법을 적용하기 위해서는 CAM에 저장된 모든 엔트리가 비트 단위로 마스킹 가능하도록 하는 것이다. 그러나 이것은 CAM 셀에 더 많은 트랜지스터를 요구하게되고, CAM 집적도를 떨어뜨리며 결과적으로 큰 라우팅 테이블을 처리하기 위해서는 더 많은 CAM을 사용하여야 하는 결과를 초래한다.

캐쉬를 이용하는 방법은 IP 주소를 캐쉬에 미리 저장하는 기법으로 평균 검색속도를 향상시킬 수 있으나 백본 라우터[1]에서는 낮은 적중률(Hit ratio) 때문에 적용하기 어렵다. [6]에서는 소규모 IP주소 캐쉬를 이용해서 포워딩 검색 속도를 최소 65% 향상시킬 수 있다고 보고하고 있다. 그러나 지금의 인터넷 환경에서 대규모 트래픽 응집도와 호스트 숫자를 고려하면 일정수준 이상의 캐쉬 적중률을 유지하기 위해서는 수천 개 이상의 캐쉬 라인이 필요하게 된다. [22]는 일반적으로 소프트웨어 기반 방법에서 사용되는 트라이를 기반으로 링크드 리스트를 이용하는 테이블을 구성하고, 구성된 테이블을 고속의 SRAM에 저장하여 빠른 검색속도를 얻을 수 있음을 보여준다.

마지막으로 파이프라인과 병렬처리 구조[8,9]를 이용하는 방법이 있다.

2.2 프로토콜 기반 연구

이 방법에서는 MPLS[19], ATM과 같이 IP 주소검색 자체를 회피하는 기법들이 있다. ATM은 패킷 포워딩 단계에서 주소검색을 회피하는 방법으로, 호 설정(Call signaling) 과정에서 네트워크에 주소를 제공하여 얻어진 VCI를 이용하여 셀을 전달한다. VCI값을 포함한 ATM 셀들은 스위치에서 VCI값이 스위칭 테이블의 인덱스로 사용되거나 해시 함수의 키로 사용되어 셀을 라우팅 검색 없이 처리할 수 있다. 이때 패킷의 크기가 클 경우에는 ATM 셀의 53 바이트 크기를 만족시키기 위해서 IP 패킷이 조각화 되어 대량의 VCI 인덱싱이 스위치에서 발생한다. 이러한 점은 특히 대형 패킷을 지원하는 IPv6에서는 더 큰 문제가 될 수 있다.

MPLS, 태그(Tag) 스위칭[20]과 플로우(Flow) 스위칭[21]은 기본적으로 ATM 위에서 IP 패킷을 우회시키는 방법이다. 즉, 데이타 링크 계층에서 ATM을 이용하여 실제적으로 패킷 스위칭을 하는데, 이때 사용되는 플로우 식별자를 특정 IP패킷에 배당하는 별도의 프로토콜이 필요하게 된다. 마지막으로 IP 패킷 헤더를 변형하는 기법[10]이 있다. [10]은 clue정보를 IP 헤더에 포함하여 라우팅 정보를 교환함으로써 경로에 존재하는 모든 라우터가 검색할 라우팅 테이블의 범위를 점차 줄인다. 이 과정을 통해 주소검색에 필요한 부담을 패킷 이동경로상의 전체 라우터에 분산시킴으로써 라우터별 부담을 경감시킬 수 있다. 이러한 프로토콜 기반 연구들은 IP위주의 네트워크에서 타 라우터와의 연동과 확장성 문제를 해결하여야할 과제로 안고있다.

2.3 소프트웨어 기반 연구

소프트웨어를 기반으로 고속 포워딩 검색을 하기 위한 연구는 먼저 전통적인 프리픽스 트라이를 이용하는 기법으로 패트리샤 트라이(Patricia trie)[11]가 있다. 이 방법은 NetBSD 1.2 에서 사용하는 포워딩 검색 기법[16]으로, 한번에 하나의 비트를 비교 처리한다. [16]에서 제시한 검색시간은 1.44 log N으로, N은 라우팅 엔트리의 개수이므로 50,000개의 엔트리가 있을 경우 22회 이상의 메모리 접근이 필요하다. 이러한 정도의 메모리 접근 필요성은 CPU 계산속도에 비해 매우 느린 메모리 속도를 감안할 때 불충분한 속도로 판단된다. 물론 패트리샤 트라이에서 반복되는 0 이나 1의 비트열을 압축해서 표현할 수 있도록 스킵 카운트(Skip count)를

사용할 수 있으나 검색실패에 따른 백트래킹이 발생할 경우에는 오히려 더 큰 부담으로 작용할 수 있다.

이 부류의 연구에서는 이진검색 기법을 적용하는 방법도 다양하게 시도되고 있다. [13]은 프리픽스 검색에 이진검색 기법을 적용할 수 있도록 프리픽스 길이를 단일하게 확장하고, 캐쉬의 워드 크기를 고려하여 이진검색을 최대 6-way 검색으로 변경할 수 있는 방법을 제시하고 있다. [14]는 프리픽스를 길이 순서대로 다수의 이진검색 테이블로 구성하고, 각 테이블을 트라이로 구성한다. [15]는 라우팅 테이블을 프리픽스 길이별로 해시 테이블에 저장한 후 프리픽스 길이를 키로 이진검색을 함으로써 해시 테이블 검색횟수를 감소시키는 방법을 제시하고 있다.

[17]은 트라이를 압축하는 방법으로 프리픽스를 이진 트라이로 표현한 후 중복된 비트 값을 제거하는 방법으로 트라이 레벨을 압축한다. 압축된 이진 트라이는 자식 노드가 두 개 이상인 다중(Multiway) 트라이로 변환하여 트라이 레벨을 추가로 압축하는 LC-trie 기법이다. [18]은 프리픽스 트라이를 두 개로 분리하여 프리픽스 트라이의 앞, 뒷부분에서 중복되는 비트열을 공통적으로 표현함으로써 트라이에 사용되는 메모리 크기를 줄일 수 있다.

Mikael의 포워딩 테이블 연구[12]는 라우팅 테이블을 일반적인 펜티엄 프로세서의 L2 캐쉬에 저장할 수 있는 작은 크기로 압축하여 기가비트 검색속도를 얻고자하는 연구이다. 이 연구에서는 먼저 라우팅 테이블을 완전이진 트라이로 표현하고, 트라이 깊이 16, 24, 32를 기준으로 각각 레벨 1, 2, 3의 세 개의 구역으로 구분한다. 먼저 16비트 길이의 레벨 1은 리프노드가 1~64K개로, 트라이를 레벨 16에서 절단한 것과 같다. 레벨 16의 각 노드에는 노드의 상태별로 한 비트씩을 할당하여 비트 벡터(bit vector)를 구성한다. 참고로 이진 트라이에서 추출된 비트 벡터의 각 비트는 IP주소 중에서 상위 16비트를 0부터 차례대로 늘여놓은 순서와 일치하는 의미를 갖는다. 따라서 비트 벡터에서 IP주소의 상위부분에 해당하는 비트를 찾기 위해서는 IP주소의 상위 16비트가 인덱스로 사용된다. 비트 벡터의 각 비트 값은 아래와 같이 노드의 상태에 따라 0 또는 1이 할당된다.

○ 루트 헤더(1): 프리픽스의 길이가 16비트 이상인 경우 즉, 트라이가 레벨 17이하로 연결되어있는 경우

○ 제뉴인 헤드(1): 레벨 16 또는 상위 레벨에 리프노드가 존재하는 경우

○ 영(0): 레벨 16 보다 상위 레벨에 있는 리프노드에 의해 대표되는 범위에 속한 노드

비트 벡터의 루트 헤더와 제뉴인 헤더는 각각 레벨 16이하의 트라이를 구조를 표현한 청크(chunk)에 대한 포인터와 라우팅 프리픽스 포인터 정보를 갖는데, 이 정보들은 16비트 길이의 포인터스(pointers) 테이블에 14비트 길이로 저장되며 나머지 2비트는 각 헤더 정보의 종류를 표시한다.

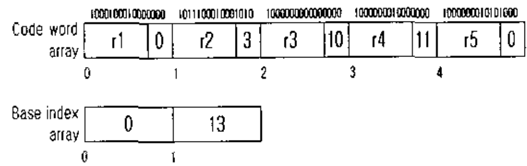


그림 1 비트-마스크에 대응하는 코드워드 및 베이스 인덱스 배열구조

IP주소의 상위 16비트 검색에 필요한 자료구조는 그림 1과 같다. 비트 벡터는 16비트의 비트-마스크로 분할되어 각 16비트에 속한 루트 헤더와 제뉴인 헤더의 개수 즉, 비트 값 1의 개수를 저장하는 코드워드(code word) 배열로 구성된다. 코드워드는 16비트 중에서 하위 6비트에 누적된 비트 값 1의 개수를 오프셋으로 저장한다. 6비트 길이는 2^6=64개의 비트 값 1의 개수를 기록할 수 있으므로 네 개의 코드 워드마다 오버플로우가 발생할 수 있다. 그러므로 네 번째 코드워드의 오프셋은 베이스 인덱스 배열에 저장하고, 다섯 번째 코드 워드는 초기화된다. 코드워드의 원소의 개수가 2^{12}개이고, 베이스 인덱스 배열은 네 개의 코드워드를 누적하고 있으므로 코드 워드 배열과 베이스 인덱스 배열에 대한 인덱스는 IP주소의 상위 12비트와 10비트가 된다. 따라서 IP주소 16비트에 대해 레벨 1을 검색하기 위해서는 나머지 4비트에 대한 정보가 필요하게된다. 이 정보를 제공하기 위해서는 레벨이 4인 트라이에서 가능한 리프노드의 조합의 가짓수를 알 필요가 있다. a(n)이 길이가 2^n인 모든 비트-마스크 조합의 개수(0 값으로 만 구성된 비트-마스크는 제외)라고 할 때 4비트 주소에 대해 가능한 비트-마스크(리프노드)의 조합은 다음과 같이 정의될 수 있다.

$$a(0) = 1,\ a(n) = 1 + a(n-1)2$$

따라서 4비트로 구성되는 16비트 길이의 비트-마스크의 가능한 조합의 개수는 a(4) +1=678이 된다. 1이 더해진 이유는 0으로만 구성된 비트열의 경우를 표시하기 위함이다. 이 연구에서는 678가지의 비트-마스크를 초

기에 맵 테이블(maptable)로 구성하고, 코드워드를 만들 때 16비트 패턴에 따라 일치하는 엔트리의 맵 테이블 인덱스를 코드워드의 상위 10비트(r1, r2,,)에 기록한다. 따라서 맵 테이블 인덱싱은 코드워드의 상위 10비트 값과 IP주소의 상의 16비트 중 마지막 4비트의 값이 동시에 사용되는 이차원 배열 인덱싱이 된다. 이제 IP주소의 상위 16비트를 검색하는 방법은 상위 10비트로 베이스 인덱스 배열을 참조하여 누적된 비트 오프셋(a)을 구하고, 다시 상위 12비트로 코드워드를 참조하여 2비트에 대한 오프셋(b)을 구한다. 마지막으로 나머지 4비트 값과 코드워드의 상위 10비트 값을 인덱스로 맵 테이블을 참조하여 나머지 오프셋(c)을 구하고, (a)+(b)+(c)로 구한 값을 포인터스 테이블에 대한 인덱스로 사용한다. 포인테스 테이블의 값은 하위 레벨의 청크 포인터 또는 라우팅 프리픽스 포인터이므로, 이 두 가지를 구분하여 IP주소 32비트에 대한 검색을 종료하거나 하위 청크에 대한 검색을 계속한다.

레벨 n의 청크는 n-1 레벨의 모든 루트헤더에 대해 생성되며, 각 청크는 8비트 깊이를 담당하므로 최대 256개의 헤더가 있다. 청크는 포함되는 헤더의 개수에 따라 세 가지 구성 방법이 사용된다. 먼저 헤더의 개수가 1~8개인 경우에는 8비트 배열과 16비트 포인터 테이블이 사용되고, 헤더가 9~64개인 경우에는 레벨 1과 같으나 베이스 인덱스 배열은 사용하지 않는다. 헤더가 65~256개인 경우에는 레벨 1과 같다. Mikael의 연구는 펜티엄 프로 200 MHz와 Alpha 21164 333 MHz에서 성능을 실험하였다. 두 개의 플랫폼에서 검색성능은 최악의 경우 각각 505ns, 444ns가 기록되었다.

3. 비트-맵 트라이

2장에서 기술한 기존의 소프트웨어 기반 고속 포워딩 검색 연구들은 다양한 자료구조와 검색방법을 사용한다. Mikael의 포워딩 테이블 연구[12]는 비트 벡터 테이블을 사용하는 연구로 프리픽스를 완전이진 트라이를 구성하면 리프노드의 순서가 가능한 모든 라우팅 프리픽스의 순서가 된다는 점을 이용한다. 이 연구에서는 트라이의 노드에 한 비트를 할당하여 만든 비트-마스크를 비트 벡터로 구성하고, 비트-벡터에서 2장에서 기술한 바와 같이 루트 헤더와 제뉴인 헤더의 개수 즉, 프리픽스의 라우팅 테이블 내부에서의 순서를 기록하기 위해 별도의 코드워드 배열과 베이스 인덱스 배열을 작성한다. 또한 IP주소 상위 16비트의 끝 부분 4비트의 패턴 검색을 위해 맵 테이블을 작성하여 IP주소 상위 16비트에 대한 검색을 한다. 검색의 결과에 따라 IP주소의 하위 16비트 정보를 갖고있는 청크에 대한 검색 필요성을 판단한다. 이 방법은 레벨 1의 비트 벡터를 구성할 때 레벨을 16으로 고정함으로써 비트 벡터의 길이가 64K가 되고 여기에는 대량의 0 비트가 삽입되어 메모리를 낭비할 수 있으며, 비트 벡터에 연결되는 추가적인 자료구조들이 필요한 단점이 있다.

본 논문은 [12]의 연구에서 비트 벡터 이외의 추가적인 자료구조를 만들지 않으며, 비트 벡터를 구성하는 과정에서도 트라이 레벨을 8로 분할하여 0 비트만으로 구성되는 비트열을 제외함으로써 다루기 쉽고, 메모리 소요량이 작으며, 빠른 포워딩 검색을 할 수 있음에 기초하여 제안되었다.

3.1 비트-맵 트라이의 자료구조

비트-맵 트라이는 포워딩 테이블의 크기를 펜티엄 프로세서의 L2 캐쉬에 저장할 수 있는 작은 크기로 압축·저장함으로써 빠른 포워딩 검색속도를 얻을 수 있다. 트라이[23]는 저장공간의 절감을 목적으로 프리픽스(Prefix) 부분을 상호 공유하는 문자열들을 표현하는데 적합한 특별한 형식의 트리 이다. 라우팅 테이블의 프리픽스 엔트리가 비트열로 표현될 수 있고 많은 프리픽스가 서로 중복되어 있기 때문에 트라이는 라우팅 테이블의 크기를 압축하여 포워딩 테이블을 구성할 때 사용할 수 있는 좋은 자료구조이다.

그림 2는 전형적인 라우팅 테이블과 그에 상응하는 트라이 표현을 보여주고 있다. 트라이의 가지(Edge)는 비트열에서 해당 비트의 값(0: 좌측 가지, 1: 우측 가지)을 표현하므로 IP주소의 특정한 부분[1]을 표시하고, 노드는 가지의 방향에 따라 발생한 비트열을 나타낸다. 특히, 검은색 노드는 노드가 의미하는 비트에 라우팅 프리픽스가 연결되어 있음을 의미하며, 본 논문에서는 프리픽스 노드라고 부른다. 그림 2에서 라우팅 프리픽스 "8*"에 해당하는 검은색 프리픽스 노드는 최상위 노드로부터 프리픽스 '8'에 대한 비트열 "00001000"을 따라 하향검색과정을 거쳐서 도달한 노드이므로 라우팅 테이블의 "8*" 프리픽스에 대한 포인터 정보 0을 포함한다.

트라이를 이용한 포워딩 검색과정의 예로 IP주소 "8.3.0.0"을 그림 2의 트라이에서 검색하면, 먼저 첫 번째 8 비트 값(IP주소[0:7]) 8(비트열: 0000 1000)에 대응하는 "8*"의 검은색 노드까지 탐색할 수 있으며, 두 번째 8 비트 값(IP주소[8:15]) 3(비트열: 0000 0011)에 대해서 검색할 때에는 트라이 깊이 16까지 탐색하려 하

1) 하나의 가지는 IP주소의 비트열 중에서 비트 하나를 의미하므로, 32 비트 IP주소의 경우 트라이의 깊이(Depth)는 최대 32가 된다.

지만 비트열대로 탐색할 수 있는 가지가 없으므로 검색에서 실패한다. 결과적으로 LPM 원칙에 의해 백-트래킹(Back-tracking)이 발생하여 검색의 중간과정에 지나쳐온 "8*"의 검은색 노드가 "8.3.0.0"을 처리할 수 있는 라우팅 프리픽스로 선택된다. 즉, 그림 2의 라우팅 테이블에서 "8*/8" 엔트리는 8 비트 프리픽스 값 8이 일치하는 IP주소에 대해서 나머지 24 비트 범위를 대표하는 프리픽스 엔트리가 되고, "8.1*/16" 엔트리는 16 비트 프리픽스 "8.1"이 일치하는 모든 IP주소에 대해서 나머지 16 비트 범위를 대표하는 프리픽스 엔트리로 사용된다.

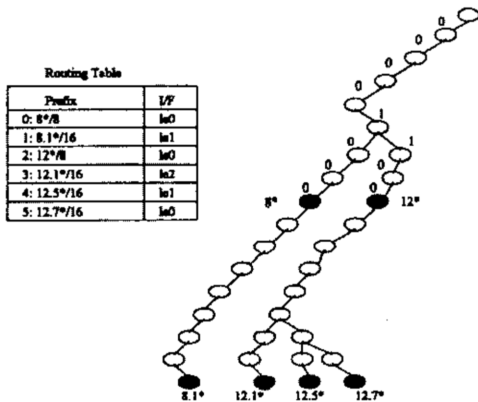


그림 2 전형적인 라우팅 테이블과 트라이 표현

32 비트 IP주소에서 표현 가능한 모든 주소를 그림 2와 같은 트라이로 표현하면 깊이는 32, 리프노드의 개수는 2^{32}개가된다. 이때 트라이의 모든 리프노드를 좌측부터 순서대로 펼쳐놓으면 모든 가능한 IP주소 값을 0부터 일렬로 늘어놓은 결과가 된다. 이러한 원리를 이용하여 트라이를 포워딩 테이블로 사용하기 위해서는 트라이 구조를 링크-리스트(Linked-list)나 배열로 표현하여야 한다. 하지만 이럴 경우 필요한 메모리의 크기가 너무 커지므로 리프노드에 연결된 라우팅 엔트리 포인터 정보의 존재여부를 하나의 비트로 표현하는 비트열을 구성하면 필요한 메모리의 크기를 크게 줄일 수 있다. 또한 비트열을 다양한 깊이에서 구성함으로써 0으로만 구성된 비트열을 제외할 수 있으며 메모리 크기도 줄일 수 있다.

트라이에서 비트열을 만드는 방법은 비트열을 만드는 깊이와 프리픽스 노드의 깊이가 다른 경우와 같은 경우의 두 가지가 있다. 먼저 깊이가 같을 때 프리픽스 노드에는 비트 1을, 공백 노드에는 0을 배당한다. 그리고 깊이가 다를 때 즉, 그림 3에서 비트열 구성 깊이는 N, 프리픽스 노드 깊이는 N-2일 때에는 1~4번째의 점선노드와 같이 프리픽스 노드로부터 깊이 N까지 가상의 자식노드를 만들고, 프리픽스 노드로 대표되는 네 개의 가상노드 중에서 좌측에서 첫 번째 노드, 노드 0에는 비트 값 1을 할당하고, 나머지 노드는 0을 할당한다. 비트 6과 12는 자식노드의 존재를 표시하기 위해 비트 값 1을 할당하였다. 이러한 포인터 압축방법은 [12]에서 트라이 깊이 16의 리프노드 마다 비트를 배당하여 비트 벡터(Bit vector)로 표현한 것과 비슷하다. 본 논문에서는 이 비트 벡터 개념을 적용하여 만든 비트열을 비트-맵(bit-map)으로 부른다.

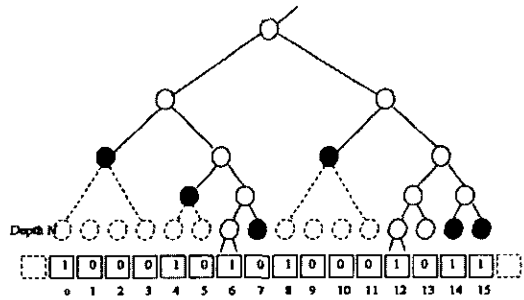


그림 3 트라이 구조에 대한 비트열 표기 방법

이렇게 구성된 비트열은 앞에서 기술한바와 같이 모든 가능한 라우팅 엔트리를 0부터 늘여놓은 형태가 되므로 첫 번째 비트부터 특정한 위치의 비트 사이에 있는 비트 값 1의 개수는 바로 특정 비트가 의미하는 비트열의 라우팅 프리픽스 인덱스가 된다. 물론 우리가 사용하는 라우팅 테이블이 2^{32}개의 라우팅 프리픽스를 갖는 것은 아니므로 트라이 또한 2^{32}개의 리프노드를 갖지는 않는다. 더구나 라우팅 프리픽스의 길이는 2비트부터 32비트까지 다양한 길이로 구성되어 있으므로 깊이가 반드시 32까지 내려가지도 않는다. 따라서 프리픽스의 길이가 N일 때에는 트라이의 노드 깊이도 N이 되며, 그 프리픽스 노드는 2^{32}개의 리프노드 중에서 하위의 $2^{(32-N)}$개의 노드를 대표하는 프리픽스가 된다.

프리픽스의 길이가 다양하게 구성되고 비트 값 1의 개수가 라우팅 프리픽스 인덱스가 되는 상태에서는 그림 2의 트라이를 그대로 사용할 수 없고 그림 4와 같이 완전이진 트라이로 확장하여야 한다. 완전이진 트라이는 자식노드가 두 개이거나 없는 트라이이다. 트라이를 완전이진 트라이로 확장해야 하는 이유는 트라이에서 구성된 비트열을 이용하여 포워딩 검색을 할 때 비트 값 1의 개수가 갖는 의미의 왜곡을 방지하기 위해서이다.

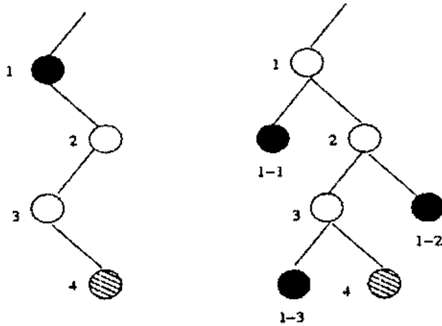


그림 4 완전이진 트라이 확장

그림 4의 좌측 트라이는 확장 이전의 트라이로써 그림 3과 같은 방법으로 비트열을 구성하면 "1000 0000"이 된다. 이 비트열에 프리픽스 "7*/4"를 검색하면 비트값 7은 비트열에서 7번째 비트를 포인트하고, 처음부터 일곱 번째 비트까지의 비트 값 1의 개수는 1이므로 라우팅 프리픽스 인덱스는 1이 된다. 인덱스 1은 결국 1번 프리픽스 노드 때문에 발생한 것으로 LPM에 의해 1번 노드가 프리픽스 "7*/4"를 대표한다는 것을 의미한다. 그러나 만일 노드 4가 프리픽스 노드라면 구성된 비트열은 "1000 0100"이 되고, 프리픽스 "5*/4" 검색결과는 2가 된다. 그러나 프리픽스 "7*/4"를 검색할 때에도 인덱스가 2가 되어 잘못된 인덱스를 갖게된다. 이러한 왜곡현상은 노드 4의 비트 값 1이 비트열에 삽입되어 좌측으로부터 비트 값 1의 개수를 계산하는 LPM 해결과정을 왜곡하기 때문이다.

이러한 문제는 그림 4의 우측과 같이 트라이를 완전이진 트라이로 확장함으로써 해결할 수 있다. 트라이 확장은 모든 자식노드를 이진 노드로 변경하고, 만일 이진 노드 확장이 발생한 노드가 프리픽스 노드일 때는 프리픽스를 자식노드로 이관한다. 그림 4의 우측노드에서 노드 1이 이진 노드로 확장되고, 인덱스가 1-1, 1-2, 1-3 노드로 이관된 예를 볼 수 있다. 완전이진 트라이로 확장된 후 구성된 비트열은 "1000 1110"으로 프리픽스 "7*/4"의 검색결과는 4가 되고, "5*/4"의 검색결과는 3이 되어 서로 다른 값을 구할 수 있다. 이때 서로 다른 범위의 노드 즉, 노드 1-1, 1-2, 1-3이 같은 프리픽스 인덱스를 참조하여야 하므로 라우팅 프리픽스 변환 테이블이 필요하게된다. 참고로 포워딩 검색에서 최장 프리픽스 검색 기준에 의해 엔트리를 검색하는 문제는 구간 집합 참여자(Interval set membership)[24] 문제와 유사하다.

완전이진 트라이로 확장된 트라이에서 그림 3과 같이 리프노드에 비트 값을 할당하여 비트열을 구성하면 트라이 구조를 표현하기 위해 필요한 포인터가 필요 없어지므로 메모리 공간을 획기적으로 줄일 수 있다. 그러나 만일 트라이 깊이 32에서 비트열을 구성한다면 비트열의 길이가 2^{32}가 되어 많은 메모리 공간을 사용하게된다. 그러므로 앞에서 기술한바와 같이 트라이에서 노드가 존재하는 부분만을 넓이-우선 탐색순서로 비트열을 구성함으로써 비트열의 길이를 줄일 수 있다. 또한 트라이를 비트열로 표현할 때에는 노드의 부모-자식노드간 구조를 표현하는 자식 비트열과 라우팅 엔트리 인덱스를 표현하는 프리픽스 비트열로 분리하여 별도로 구성하여야 한다. 따라서 제안된 비트-맵 트라이는 자식 비트-맵 배열과 프리픽스 비트-맵 배열로 구성된다.

그림 5는 그림 2에서 사용된 라우팅 테이블을 완전이진 트라이로 확장한 모양과 확장된 트라이 구조를 표현하는 자식 비트열을 보여준다. 자식 비트열은 노드에서 자식노드를 참조하는 가지의 존재 유무에 따라 1 또는 0 값이 할당된다. 비트열들은 트라이의 일정 깊이마다 구성되어 자식 비트-맵(Child bit-map) 배열의 원소(노드)가 된다. 이때 모든 노드는 넓이-우선 탐색순서로 수집되고, 비트열의 길이는 비트열이 구성되는 깊이에 따라 달라진다. 본 논문에서는 트라이의 깊이가 8의 배수가 되는 깊이에서 비트열을 구성하였으므로 비트열의 길이는 2^8 즉, 256 비트가 된다. 비트열을 구성하는 깊이가 깊어지면 비트열의 길이가 길어져서 0으로 채워지는 길이가 길어지는 메모리 낭비가 발생할 수 있으며, 너무 짧을 때에는 노드의 개수가 많아져서 검색시간이 길어진다.

그림 5의 트라이에서 자식 비트-맵 배열을 만들기 위한 비트열 구성과정은 다음과 같다.

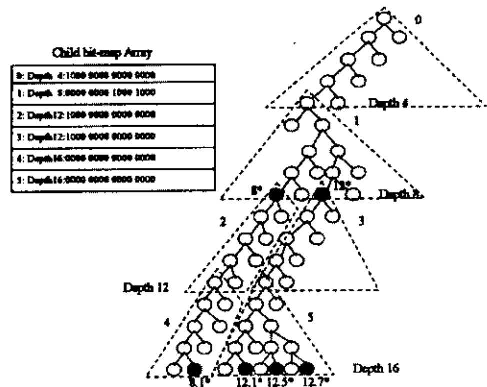


그림 5 트라이의 자식 비트-맵 노드 표현 예제

참고로 그림 5의 자식 비트-맵 배열 구성은 설명의 편의를 위해 트라이의 깊이가 4의 배수가 되는 위치에서 비트열을 구성한다. 먼저 비트열의 구성순서는 넓이-우선 탐색순서대로 수집되므로 트라이의 점선 삼각형 번호대로 자식 비트-맵 배열에 저장된다. 점선 삼각형의 깊이가 4이므로 삼각형에 속하는 리프노드의 개수는 16개가되고, 점선 삼각형 0번의 리프노드에서 자식노드를 가진 노드가 비트 값 1을 할당받는다. 또한 비트 값 1이 할당된 리프노드는 하위 자식노드에 대한 연결구조를 유지하기 위해 점선 삼각형 1번처럼 하위의 다른 점선 삼각형의 루트노드가 된다. 점선 삼각형 1번은 두 개의 자식노드 연결노드가 있어 두 개의 비트 값 1을 갖는 비트열로 표현된다.

그림 5의 자식 비트-맵 배열에서 트라이를 탐색하는 방법은 배열 노드의 비트 값 1의 개수가 그림 5의 트라이에 표시된 점선 삼각형의 순서 즉, 배열 노드의 인덱스 값이라는 사실을 이용한다. 다시 설명하자면 탐색할 주소의 4 비트 값은 점선 삼각형(즉, 배열 노드의 비트열)의 리프노드 위치를 의미하며, 배열의 첫 번째 노드부터 그 위치까지 존재하는 비트 값 1의 합계는 하위 점선 삼각형의 순서를 의미한다. 탐색은 배열의 첫 번째 노드, 인덱스 0에서 시작하며, 각 배열 노드가 깊이 4(루트노드부터 +4 깊이까지)에서 구성되었으므로 검색할 IP주소 값은 4비트 단위로 사용된다. 그림 5의 트라이는 최대 깊이가 16이므로 4비트 단위의 검색은 최대 4회 실시되며, 이것은 트라이를 루트노드로부터 탑-다운 방향으로 검색하는 것을 의미한다. 상세한 알고리즘은 3.2절의 검색 알고리즘에서 기술한다.

검색할 주소를 "8.1"로 할 때 첫 번째 노드를 탐색할 4 비트 값은 "8.1[0:3]" = 0이고, 이 값은 첫 번째 노드의 비트 위치 즉, 비트 인덱스이다. 그러므로 배열의 첫 번째 노드의 좌측으로부터 비트 인덱스 0까지의 1의 개수는 1개이고, 이 1 값은 두 번째 4 비트 값 "8.1[4:7]" = 8을 탐색할 배열 노드의 인덱스 값이다. 두 번째 4 비트 값 8의 탐색은 배열의 첫 번째 노드로부터 두 번째 노드, 배열 인덱스 1 노드에서 8번째 비트 위치사이에 있는 비트 값 1의 개수는 2가 된다. 이러한 과정은 "8.1[8:11]", "8.1[12:15]"까지 반복되며, 최종적으로 "8.1*" 프리픽스 노드를 탐색하게 된다. 하지만 "8.1*" 프리픽스 노드에 자식노드가 연결되어 있지 않으므로 더 이상 하위로 탐색할 수 없다는 결과를 얻게된다.

자식 비트-맵 배열에서 자식 노드 탐색을 위해 각 노드에 있는 비트 값 1의 개수를 계산할 때 비트 시프트(Shift) 연산이 반복적으로 발생하는 것은 상당한 오버헤드가 되므로 일정한 길이로 비트열을 구분하여 각 길이별로 비트 값 1의 개수를 함께 저장함으로써 자식 비트-맵 탐색시간을 줄일 수 있다. 그림 6은 16 비트 비트열을 4개의 4 비트 구간으로 나누어 좌측 3개의 구간에 있는 비트 1의 개수를 함께 저장하도록 수정된 자식 비트-맵 배열을 보여주고 있다. 그림 6의 *tc* 필드는 자식 비트-맵 배열에서 배열의 첫 비트부터 자신을 제외한 모든 이전 노드에 속한 비트 값 1의 개수이고, *c1,c2,c3*은 해당 노드에서 각 4 비트 구간[2]에 속한 비트 값 1의 개수이다. 비트 값 1의 개수를 저장하기 위해 추가로 필요한 공간은 노드 당 다섯 바이트 정도로 약 5만개의 라우팅 엔트리에서 35K~40K 바이트가 소요되는데, 이 크기는 포워딩 테이블을 캐쉬에 저장할 때 부담이 되지 않는다.

Child bit-map Array

	tc	c1	c2	c3	bit-map
Depth4:	0	1	0	0	1000 0000 0000 0000
Depth8:	1	0	0	1	0000 0000 1000 1000
Depth12:	3	1	0	0	1000 0000 0000 0000
Depth12:	4	1	0	0	1000 0000 0000 0000
Depth16:	5	0	0	0	0000 0000 0000 0000
Depth16:	5	0	0	0	0000 0000 0000 0000

그림 6 비트 값 1의 구간별 개수를 함께 기록한 자식 비트-맵 배열

그림 5의 트라이에서 프리픽스 노드에 대한 정보를 담은 프리픽스 비트-맵 배열의 구성은 앞에서 기술한 자식 비트-맵 배열을 만드는 방법과 동일하다. 다만 비트 값을 할당할 때 프리픽스 비트에는 1을 할당한다. 그림 5의 트라이에서 구성한 프리픽스 비트-맵 배열의 예는 그림 7에서 볼 수 있다.

Prefix bit-map Array

	tp	p1	p2	p3	bit-map
Depth4:	0	0	0	0	0000 0000 0000 0000
Depth8:	0	0	0	1	0000 0000 1000 1000
Depth12:	1	0	0	0	0000 0000 0000 0000
Depth12:	1	0	0	0	0000 0000 0000 0000
Depth16:	1	1	0	0	0100 0000 0000 0000
Depth16:	2	1	2	0	0100 0101 0000 0000

그림 7 프리픽스 비트-맵 배열

2) 실제 자식 비트-맵 배열에서는 트라이 깊이 8마다 비트열이 만들어지고, 비트열의 길이가 256 비트가 되므로 각 구간의 길이는 64 비트가 된다.

자식 비트-맵과 마찬가지로 프리픽스 비트-맵 배열에서도 각 비트열 구간의 비트 값 1의 개수를 함께 저장함으로써 검색속도를 증가시킬 수 있다.

그림 8은 본 논문에서 제안한 비트-맵 트라이와 비트-맵 테이블의 구성 예제로, 그림 8 a) 비트-맵 트라이는 라우팅 테이블을 표현하는 트라이가 삼각형 모양의 부 트라이의 집합으로 분할되고 있음을 보여주고 있다. 트라이 깊이 8의 배수에서 구성된 각각의 부 트라이들은 길이가 256 비트인 비트열로써 비트-맵 테이블의 노드가 되며, 각 노드는 자식 비트-맵 노드와 프리픽스 비트-맵 노드로 구성된다. 그림 8 a) 비트-맵 트라이에서 최상위 부 트라이의 256개의 리프노드가 점선 사각형으로 표시되어 있는데, 이 사각형 내부의 256개 리프노드가 두 개의 256 비트의 비트열로 각각 프리픽스 비트-맵 노드와 자식 비트-맵 노드로 표현되며, 자식 비트 값에 따라 최대 256개의 하위의 부 트라이를 가짐으로써 트라이의 자식노드 구조에 대한 정보를 표현한다.

그림 8 b) 비트-맵 테이블은 별도의 배열로 구성되었던 프리픽스 비트-맵 배열과 자식 비트-맵 배열의 각 노드가 통합되어 하나의 비트-맵 테이블 노드를 구성하는 것을 보여준다. 또한 각 비트-맵 테이블 노드는 비트 1의 개수를 빠르게 계산하기 위해 전체 이전 노드의 1의 개수의 합을 나타내는 *tp*와 *tc* 필드, 256 비트 노드를 4개의 구간으로 분리하여 64 비트 구간 세 개의 비트 1의 개수를 나타내는 *p1, p2, p3* 및 *c1, c2, c3* 필드를 포함한다.

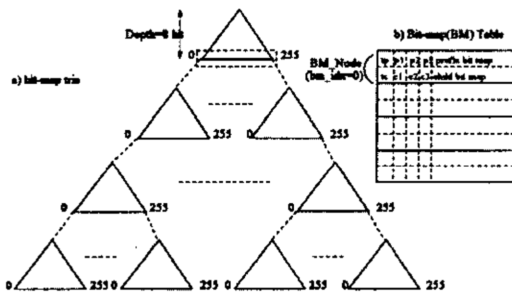


그림 8 비트-맵 트라이와 비트-맵 테이블

3.2 검색 알고리즘

본 논문에서 제안된 비트-맵 트라이를 이용하는 검색 알고리즘은 두 가지가 있다. 첫 번째 검색 알고리즘은 본 논문의 표준 검색 알고리즘으로 비트-맵 테이블을 루트노드로부터 탑-다운 검색을 한다. 이 방법에서는 입력된 IP주소 32비트를 8비트씩 나누어서 최대 4번의 비교 연산을 실시한다. 두 번째 검색 방법인 이진검색 알고리즘은 탑-다운 검색을 하는 표준 검색 알고리즘에 비해 중간 레벨의 노드로부터 검색을 시작할 수 있는 것이 장점이다. 루트노드로부터 검색을 시작하지 않을 수 있다는 것은 128비트 길이의 IPv6 주소를 사용하는 라우팅 테이블을 비트-맵으로 구성했을 때 유용하다. 즉 32비트 IPv4 주소에서는 최대 4번의 비교 연산이 128비트 길이에서는 최대 16번의 연산이 필요하기 때문에 중간 레벨의 노드에서 검색을 시작함으로써 필요한 연산의 회수를 감소시킬 수 있다. 여기에는 약간의 추가적인 메모리 공간이 필요한 단점이 있다. 참고로 실험에 사용할 수 있는 수천~수만 개의 엔트리를 가진 IPv6 라우팅 테이블이 아직 존재하지 않으므로 본 논문에서는 이진검색 알고리즘을 IPv4 라우팅 테이블에 적용하여 성능을 측정하였다. 또한 이진검색 알고리즘이 128비트 길이의 주소검색 회수를 16번에서 8번으로 감소하여도 8번의 비교 연산은 너무 많은 회수이므로 추가적으로 트라이의 레벨을 압축할 수 있는 기법이 사용되어야 할 것으로 판단된다.

3.2.1 표준 검색 알고리즘

비트-맵 테이블을 이용한 표준 검색 알고리즘은 먼저 검색할 IP주소 값에 따라 비트-맵 테이블의 자식 비트-맵 노드를 검색한다. 최장 프리픽스 매칭 원칙에 따라 자식노드가 연결되어있는 리프노드까지 검색하고, 더 이상 자식노드가 연결되어있지 않은 리프노드에 도착하면 그림 8의 같은 MB_NODE에 속하는 프리픽스 노드에서 동일한 비트위치까지의 비트 1의 개수를 계산하고, 계산된 1의 개수를 완전이진 트라이 확장 때 만든 변환 테이블의 인덱스로 이용해서 라우팅 프리픽스 인덱스를 구한다. 첫 번째 자식 비트-맵에서 자식노드 링크가 없고, 동일한 위치의 프리픽스 노드의 비트 값이 1이 아니면 해당 IP 주소에 대한 검색은 실패가 되고, 일반적으로 디폴트 엔트리를 선택하게된다.

자식노드 검색에서 사용할 첫 번째 *BM* 테이블 노드 인덱스(*bm_idx*)는 초기 값이 0이다. 즉 트라이에서 최상위 삼각형이 첫 번째 검색 대상이다. 그림 9의 라인 3은 검색할 IP주소에서 자식노드를 검색할 때 사용할 8 비트 인덱스 값(*addr*)을 할당한다. *addr*의 의미는 2.1절에서 설명한바와 같이 자식 비트-맵 노드의 비트 인덱스이다. 즉 그림 8의 삼각형에 속한 256개의 리프노드 순서를 의미한다. 라인 4~11은 *addr* 비트 인덱스 값에 대해 자식노드가 있는지 검사하여 자식노드가 있을 경우에 라인 8에서 *bm_idx*에 그 자식노드가 속한 곳의 주소 값을 기록한다. *bm_idx* 값은 자식 비트-맵의 처음

```
1: Let Bit-Map table BM and an index bm_idx:=0, and
2: routing entry pointer ptr:=default routing entry
3: for (addr=IP_Address[m*8:m*8+7], m=0,1,2,3) {
4:    get BM_Node:=BM(bm_idx);
5:    get byte_idx:=addr / 8;
6:    get bit_idx:=addr mod 8;
7:    if (BM_node.childBit(byte_idx)<<bit_idx & 0x80) {
8:         set bm_idx:=number of all child bit 1; and
9:    }
10:   else
11:        set ptr:=number of all prefix bit 1;
12: }
13: return pointers[ptr];
```

그림 9 비트-맵 테이블을 이용한 검색 알고리즘

부터 *addr* 비트 인덱스사이에 있는 비트 값 1의 개수이며, 다음 8 비트 값으로 검색할 비트-맵 테이블의 인덱스 값이다. 즉, 그림 8에서 다음에 검색할 삼각형의 인덱스 값이다. 라인 7, 13에서 *BM_Node.childBit*는 자식 비트-맵 노드를 의미한다. 라인 11에서는 더 이상 검색할 자식노드가 없거나 IP주소의 네 번째(m=3) 8 비트 값까지 검색이 된 경우에 라우팅 프리픽스 인덱스 *ptr*을 계산한다

검색의 결과로 구해진 *ptr*는 라우팅 테이블 엔트리의 인덱스가 된다. 그러나 완전이진 트라이 변환 때문에 비트-맵 테이블의 비트 값 1의 개수는 라우팅 테이블의 엔트리 개수보다 많으므로 구해진 인덱스 *ptr*로 직접 라우팅 테이블을 읽을 수는 없다. 비트-맵 테이블의 비트 값 1의 개수는 확장된 트라이 기반 인덱스이므로 확장되기 전의 트라이 인덱스로 변환시키는 별도의 테이블(*pointer*)을 사용한다. *pointer* 테이블 사용으로 포워딩 검색에 필요한 메모리 읽기 회수가 1회 추가로 발생 하지만 pointer 테이블의 크기는 50,000개 엔트리를 가진 라우팅 테이블에서 약 100K 바이트로 메모리 읽기 대신 캐쉬 읽기가 발생할 것이다.

3.2.2 이진 검색 알고리즘

비트-맵 테이블을 이진검색(Binary search)하는 방법은 그림 10의 알고리즘과 같다. 본 논문에서 이진 검색의 의미는 트라이를 최상위 노드부터 검색하지 않고 중간 노드부터 검색하는 것이다. 2.1절의 비트-맵 자료구조에서 기술한바와 같이 최대 깊이 32의 트라이는 깊이 8의 배수마다 부 트라이가 구성되고, 각 부 트라이는 비트-맵 테이블의 노드로 저장된다. 이진검색은 최상위의 부 트라이에서 검색을 시작하지 않고, 깊이 8또는 16에서 형성된 부 트라이에서 검색을 시작함으로써 포워딩 검색시간을 단축할 수 있다. 이진검색은 IPv6의 128 비트 주소를 트라이로 표현했을 때 발생하는 깊은 트라이 깊이 문제를 해결할 수 있는 출발점을 제시하기도 한다.

```
1: Let Bit-Map table BM and an index bm_idx:=0, and
2: strt_pos:=0 and
3: routing entry pointer ptr:=default routing entry
4: get b_byte_idx:=IP_Address[0:15] / 8;
5: get b_bit_idx:=IP_Address[0:15] mod 8;
6: if (pnt16_child[b_byte_idx]<<b_bit_idx & 0x80) {
7:   bm_idx:=number of all child bit 1; and
8:   set strt_pos:=2; and
9: }
10: else {
11:   set ptr:=number of all prefix bit 1;
12:   return pointers[ptr];
13: }
14: for (addr=IP_Address[m*8:m*8+7], m=strt_pos~3) {
15:   same to #4 ~ #11 lines of Algorithm 1
16: }
17: return pointers[ptr];
```

그림 10 비트-맵 테이블을 이용한 이진 검색 알고리즘

기본적으로 그림 10의 알고리즘은 라인 15와 같이 그림 9의 알고리즘과 같은 검색방법을 사용한다. 다만 라인 14에서 그림 9의 알고리즘과는 달리 라인 4~13의 이진검색 결과를 반영하여 라인 14에서 *strt_pos*를 조정하여 16보다 짧은 프리픽스의 검색 시작위치를 유동적으로 적용하고 있다. 라인 2에서는 이러한 *strt_pos*를 0으로 초기화하고, 라인 8에서는 이진노드 검색이 성공했을 때 라인 14에서 다시 짧은 프리픽스 검색위치를 변경한다.

라인 6에서 사용된 *pnt16[]*은 트라이 깊이 16에 해당하는 모든 부 트라이[3]를 좌측으로부터 차례로 자식노드 비트열, 프리픽스 비트열로 구성한 것이다. 각 비트열은 비트 1의 개수를 빠르게 계산할 수 있도록 일정한 폭마다 누적치를 함께 저장하며, *pnt16[]*의 크기는 최대 8K 바이트 메모리가 자식노드와 프리픽스를 위해 두 개가 사용된다. 이 크기는 비트-맵 테이블로 구성된 포워딩 테이블을 일반적인 펜티엄 프로세서의 L2 캐쉬에 저장할 때 별다른 영향을 주지 않는다. 참고로 알고리즘2의 이진검색 예는 트라이 깊이 32를 고려하여 실제 사용효과가 있는 트라이 깊이 16에만 이진검색을 적용한 경우이며, 16의 의미는 트라이 깊이 16을 의미한다.

라인 6은 *pnt16[]*에서 *b_byte_idx/b_bit_idx* 위치에서 자식노드의 존재를 검사하고, 자식노드 비트 값 1이 있을 때에는 라인 7에서 자식노드 인덱스 *bm_idx*를 산

3) 이 경우 부 트라이의 루트의 깊이는 8이다.

출한다. 라인 10은 깊이 16에 자식노드가 없을 경우, 같은 위치의 프리픽스 노드에서 좌측으로부터 비트 1의 개수를 계산하여 *ptr*값을 구하고, 포워딩 검색을 종료한다. 포워딩 검색이 종료되는 이유는 자식노드가 더 이상 없는 상태가 이미 확인됐으므로 하위 노드검색이 불필요하기 때문이다. 만일 라인 6과 12에서 *pnt16[]*으로 아무런 정보도 얻지 못했다면 프리픽스 길이가 16 보다 짧은 라우팅 프리픽스를 검색하여야 한다. 그림 9의 알고리즘의 절차가 적용될 것이며, 자식노드가 검출됐다면 LPM에 의해 그 이후의 나머지 비트 값에 대해서만 그림 9의 알고리즘이 적용된다.

4. 성능측정

비트-맵 테이블을 이용한 포워딩 실험은 표 1과 같이 미국 백본망의 실제 라우팅 테이블[25]을 이용하여 실시하였으며, 실험결과는 본 논문에서 제시한 비트-맵 테이블로 구현된 포워딩 테이블이 일반적인 범용 프로세서에서 기가비트급 라우터 성능을 충분히 만족시킬 수 있음을 보여주었다.

비트-맵 포워딩 테이블을 이용한 포워딩 검색실험에 사용된 운영체제와 CPU는 리눅스 커널 2.2.x에서 400 ㎒ 펜티엄Ⅱ 프로세서(L1 cache 32KByte, L2 cache 512KByte)를 이용하였다. 실험에서 사용된 라우팅 테이블의 크기는 4만8천여 개 이상의 백본급 라우터부터 수천 개의 엔트리를 가진 기업(Enterprise)급 라우터까지 다양하게 구성되어 있다. 비트-맵 포워딩 테이블의 크기는 141K 바이트부터 284K 바이트까지 다양한 크기로 생성되었다. 이 크기는 일반적인 펜티엄 CPU의 L2 캐쉬 크기에 충분히 저장될 수 있는 크기로 본 논문이 예상한 결과와 부합됨을 알 수 있다. 라우팅 테이블로부터 포워딩 테이블의 구성(Build)까지 소요된 시간은 메모리에 있는 프리픽스 트라이로부터 비트-맵 테이블을 구성하는 시간으로 약 90㎳부터 280㎳가 소요되었다. [26]에 의하면 백본급 라우터의 포워딩 테이블이 초당 1회 정도의 수정 빈도를 가짐을 감안할 때 충분히 적은 시간이라고 할 수 있다.

포워딩 검색은 네 가지 실험 시나리오를 통해 실험하였다. 첫 번째 시나리오는 표1의 라우팅 테이블을 대상으로 랜덤(Random) 함수를 통해 생성된 IP주소를 사용하여 검색을 실시하였고, 두 번째 시나리오는 표1의 라우팅 테이블을 대상으로 본 기관의 라우터 박스에서 수집한 실제 IP주소를 검색하도록 하였다. 세 번째 시나리오는 본 기관의 라우터에서 추출한 라우팅 테이블을 대상으로 비트-맵 테이블을 구성하고 랜덤함수를 통해 생성된 IP주소를 사용하였다. 마지막 시나리오는 본 기관의 라우팅 테이블과 실제 수집한 IP주소를 사용하였다. 본 기관의 라우팅 테이블은 두 개의 BGP[27] 라우터에서 각기 다른 날짜에 세 번 라우팅 테이블을 수집하였고, 동시에 하루동안 라우터를 통과한 IP 패킷정보를 수집하였다. 참고로 우리의 이해 범위 내에서는 현재의 일반적인 프로세서는 특정 데이타 블록을 L2 캐쉬에 상주시킬 수 있는 특별한 방법이 없으므로, 우리의 실험 결과는 포워딩 테이블이 L2 캐쉬와 메모리에 혼재하고 있는 상태의 결과 값으로 판단된다.

표 1 포워딩 테이블 생성 데이타

Site	Date	No. of Routing entries	Leaves	Bit-Map Table size(KB)	Build time (㎳)
MaeEast	99.12	48,290	52,858	284	290
MaeWest	99.12	29,588	32,462	247	210
PacBell	99.12	25,275	26,897	226	190
AADS	99.12	16,864	18,093	190	140
Paix	99.12	9,618	10,434	141	80

첫 번째 시나리오에 의한 실험결과는 표2에 정리하였다. 사용된 라우팅 테이블은 총5개로 모두 총 2백만 개의 패킷을 입력하였고, 총 검색시간을 패킷의 개수로 나누어 패킷별 검색시간을 산출하였다. 나머지 시나리오에서도 총 검색시간을 패킷의 개수로 나누는 방법으로 소요시간을 산출하였다. 포워딩 검색은 2.2절과 2.3절에 제시된 두 가지 검색 알고리즘별로 실험하였다. 먼저 2.2절의 표준 검색 알고리즘에 의한 패킷별 포워딩 검색시간은 라우팅 테이블의 종류별로 175ns부터 240ns의 범위에서 기록되었다. 2.3절의 이진 검색 알고리즘에 의한 검색시간은 130ns부터 175ns의 범위로 표준 검색 알고리즘에 비해 평균 27%의 성능 향상을 얻을 수 있다. 패킷의 평균 길이 512 바이트에서 5 Gbps 라우터 속도를 지원하기 위해서는 패킷 검색이 800ns 이내에 수행되어야함을 감안할 때 우리의 비트-맵 포워딩 구조는 최대

표 2 IPMA DB와 랜덤 IP 주소에 의한 실험결과

	Mae-East	Mae-West	PacBell	AADS	Paix
No. of Packet	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
Resolved ratio	22.6%	19.1%	10.3%	8.3%	5.3%
Seek Time(ns)	240	225	200	195	175
Binary Seek Time(ns)	175	170	145	140	130

31 Gbps까지 지원할 수 있는 고속 포워딩 구조로 평가할 수 있다.

두 번째 시나리오의 결과는 표 3에서 볼 수 있다. 실험에 사용된 IP주소는 본 기관에서 하루동안 수집된 실제 주소를 사용했으며, 표 2의 결과와는 다르게 검색 성공률이 거의 100%에 이른다. 그러나 검색시간은 약 2~3 배정도 더 소요되고 있음을 확인할 수 있다. 즉, 표 2와 3을 비교할 때 랜덤함수로 작성된 IP주소와 실제 IP주소의 차이가 검색 성공률과 검색시간에 일정한 영향을 미치고 있다는 것을 쉽게 확인할 수 있다.

표 3 IPMA DB와 실제 IP 주소에 의한 실험결과

	Mac-East	Mae-West	PacBell	AADS	Paix
No. of Packet	6,199,066	6,199,066	6,199,066	6,199,066	6,199,066
Resolved ratio	100%	99.2%	99.2%	99.2%	99.2%
Seek Time(ns)	674	670	672	670	501
Binary Seek Time(ns)	395	395	388	396	388

세 번째 시나리오에 의한 실험결과는 표4에서 볼 수 있다. 라우팅 테이블은 두 종류로 R1과 R2가 사용되고, 각각 3회씩 수집한 테이블이 실험에 사용되었다. 표2의 첫 번째 시나리오 결과와 비슷한 결과를 볼 수 있는데, 상대적으로 더 적은 성공률 때문에 더 높은 검색성능을 보이고 있음을 확인할 수 있다.

표 4 본 연구기관 DB와 랜덤 IP 주소에 의한 실험결과

	R1_1	R1_2	R1_3	R2_1	R2_2	R2_3
No. of Packet	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
Resolved ratio	2.8%	2.8%	2.7%	2.8%	2.8%	2.7%
Seek Time(ns)	160	165	165	160	165	165
Binary Seek Time(ns)	120	120	120	120	120	120

마지막으로 네 번째 시나리오에 의한 실험결과는 표 5에 정리되었다. 표 5에서 볼 수 있는 실험결과는 표2, 4의 결과와 많은 부분에서 틀린 점을 보여주고 있다. 먼저 포워딩 검색속도가 뚜렷하게 느려짐을 확인할 수 있다. 표준 검색 알고리즘의 검색소요 시간은 388ns부터 544ns, 이진 검색 알고리즘에서는 256ns부터 360ns사이에서 분포되었다. 또한 실험에서 입력된 패킷의 개수가 다른데, 이것은 실제 하루동안 관측된 패킷을 수집하여 사용했기 때문이다.

표 5 본 연구기관 DB와 실제 IP 주소에 의한 실험결과

	R1_1	R1_2	R1_3	R2_1	R2_2	R2_3
No. of Packet	2,262,708	6,199,066	6,184,852	12,446,452	3,088,266	2,553,654
Resolved Ratio	99.9%	100%	100%	100%	100%	100%
Seek Time(ns)	388	500	499	501	511	544
Binary Seek Time(ns)	256	364	365	362	365	360

네 가지 실험 시나리오에 의한 실험결과에서 두 번째, 네 번째 시나리오 결과와 첫 번째 및 세 번째 실험결과의 검색속도가 많은 차이를 보이는 이유는 입력된 패킷이 포워딩 검색에 의해 라우팅 엔트리를 찾은 비율인 "Resolved Ratio" 즉, 검색 성공률과 깊은 관련이 있다. 즉, 세 번째 시나리오의 검색 성공률이 100%로 앞의 두 가지 시나리오의 결과보다 월등히 높은 것이 검색시간이 증가한 이유이다. 이러한 사실은 그림 11의 그래프를 통해서도 확인할 수 있다.

그림 11은 표2의 다섯 가지 라우팅 테이블에 대해서 검색 성공률과 검색시간의 관계를 표시한 것으로 검색 성공률 곡선과 검색시간 곡선이 매우 유사한 모양으로 닮아있음을 보여준다. 비트-맵 포워딩 테이블의 검색 알고리즘은 32 비트 주소를 8 비트 단위로 분할해서 차례대로 비트-맵 테이블 노드를 검색하는데, 검색 성공률이 낮은 경우 8 비트, 16 비트 값 검색에서 실패하여 포워딩 검색이 종료되는 비율이 높아지게 된다. 반대로 검색 성공 시에는 16 비트, 24 비트 및 32 비트까지 검색하는 비율이 상대적으로 더 많아지므로 검색시간이 증가한다. 첫 번째 및 세 번째 실험에서 사용된 IP주소는 랜덤함수를 이용해서 만든 것으로 라우팅 테이블에 등록되어있지 않거나 IP주소 영역 중에서 사용하지 않는 영역의 주소들이 생성된 결과로 검색 성공률이 낮게 측정되고, 두 번째 및 네 번째 실험의 경우에는 실제 트래픽을 수집하여 IP주소를 추출하였기 때문에 거의 100% 검색성공률을 기록한 것이다. 이러한 결과를 바탕으로 볼 때 기존의 소프트웨어 기반 연구들[23,24,28]에서 기록한 검색시간은 모두 IP주소를 랜덤함수로 작성하여 사용했으므로 실제 라우터 박스에 적용할 경우 표4의 결과처럼 상당한 정도의 검색시간 증가가 발생할 것으로 생각된다.

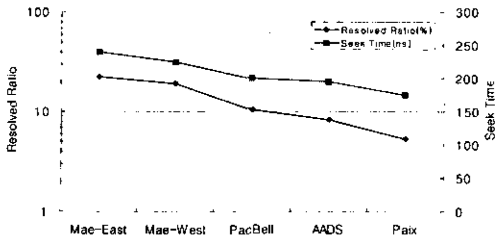


그림 11 표2의 검색성공률과 검색시간 사이의 상관관계

5. 확장성

비트-맵 테이블을 이용한 포워딩 검색은 테이블의 크기가 펜티엄 프로세서의 L2 캐쉬에 저장될 수 있는 크기로 구성된다. 따라서 라우팅 테이블 엔트리 개수의 증가가 테이블 크기의 증가로 귀결되어 비트-맵 테이블의 일부가 속도가 느린 메모리에 저장되지 않도록 관리하여야 한다.

5.1 포워딩 테이블 크기

표 1에서 가장 큰 Mae-East 데이타베이스의 약 48,000 여 개의 라우팅 엔트리에 대해서 포워딩 테이블의 크기는 284K 바이트이고 약 9,000 여 개의 엔트리를 가진 Paix의 테이블 크기는 141K 바이트이다. 두 번째 크기의 Mae-West는 약 29,000 여 개의 엔트리에 247K 바이트의 테이블 크기를 갖고 있다. 라우팅 엔트리 개수의 증가는 Paix를 기준으로 할 때 Mae-East는 5.3배, Mae-West는 3.2배 증가하고 있지만 비트-맵 테이블의 크기는 각각 2배, 1.75배를 기록하고 있다. 그 이유는 라우팅 엔트리의 개수가 충분히 커질 경우 트라이의 리프 노드 분포가 일정한 정도로 밀집(Dense) 형태를 유지하기 때문에 리프 노드 개수 증가가 테이블 크기 증가로 연결되지 않기 때문이다. 반대로 라우팅 엔트리의 개수가 작을 때는 동일한 엔트리 개수에 대해서도 프리픽스의 분포가 성긴(Sparse) 형태일 때 최악의 테이블 크기를 작성하게 되는 것이다. 따라서 Mae-East와 같은 정도로 라우팅 엔트리가 분포한다고 가정할 때 기존의 비트-맵 테이블은 약 10만여 개의 라우팅 엔트리를 지원할 수 있으며, 실제로 10만여 개 이상의 거대한 라우팅 테이블이 등장할 경우에는 트라이 레벨 압축기술 적용으로 테이블 크기 문제는 해결할 수 있다.

5.2 IPv6 확장

IPv6[28] 주소는 128 비트로 기존의 IPv4 32 비트 주소에 비해 4배의 길이로 정의되어 있다. 128 비트 주소는 IPv4의 32 비트 주소 크기 때문에 발생하는 여러 가지 문제점을 해결하기 위한 방안이지만 Gbps급 고속 라우터를 구현하기 위한 노력에는 어려움을 더해주는 결과가 되었다. 일례로 고속 라우터를 구현하기 위한 소프트웨어 기반 연구 중에서 [14,15]은 프리픽스 길이를 이용해서 각각 이진검색 테이블과 해시 테이블을 구성하는데 128 비트 주소 길이는 확장성과 검색시간에 문제점을 발생시킨다. 또한 지금 시점에서는 128 비트 주소 길이 중에서 어느 정도의 길이의 프리픽스로 주소가 배분될 것인지에 대한 완전한 합의가 도출되지 않은 상태에서 정식 IPv6 주소가 배분되고 있는 상태이며, 만일 배분된 주소가 애초의 의도와 달리 충분히 계층적·종합적으로 사용되지 않는다면 라우팅 테이블의 크기가 크게 증가할 가능성이 있다.

본 연구에서 실험한 비트-맵 트라이는 32 비트 주소에서 최대 프리픽스 길이 32를 깊이로 트라이를 구성하므로 IPv6 주소에서는 상당한 정도의 포워딩 테이블의 크기 증가가 예상된다. 그러나 트라이 레벨 압축기술을 적용하면 이러한 문제점은 거의 해결될 것이고 IPv6 주소에 대해서도 좋은 결과를 보여줄 것이다. 또한 IPv6 주소가 갖는 특성을 충분히 반영한 최적화 결과가 반영될 수 있다면 더 좋은 결과를 도출할 것이다. 그러나 최근의 IPv6 실험망인 6Bone 이나 IPv6 전용망의 라우팅 테이블은 실용적 수준의 라우팅 테이블이 존재하지 않는 이유로 이러한 연구를 진행할 수는 없는 상태이다.

6. 결 론

본 논문은 기가비트 라우터를 지원할 수 있는 포워딩 테이블을 구현하기 위한 새로운 자료구조를 제안하였다. 실험을 통해 구현된 비트-맵 자료구조는 랜덤함수에 의해 생성된 IP주소를 사용할 때 표준 검색 알고리즘에서 평균 175ns~240ns, 이진 검색 알고리즘에서는 130ns~175ns의 검색 속도로 최대 31 Gbps 성능을 보였다. 실제 IP주소를 사용할 때는 다소 느린 속도를 보였는데, 표준 및 이진 검색 알고리즘에서 각각 388ns~544ns와 256ns~360ns가 측정되어 최대 15 Gbps 속도를 지원할 수 있다. 또한 구현된 포워딩 테이블 자료구조는 약 48,000여 개의 엔트리를 가진 라우팅 테이블에 대해서 약 280K 바이트의 크기로 일반적인 펜티엄 프로세서의 L2 캐쉬에 저장될 수 있는 크기로 구현되었다.

향후의 연구는 128 비트 주소를 사용하는 IPv6 주소를 캐쉬 크기로 표현할 수 있는 트라이 레벨 압축 방법을 연구 실험하는 것이다. 또한 최근의 IPv6 주소 할당 정책과 IPv6 주소 아키텍처를 참조하여 IPv6 라우팅 테이블의 특성을 반영할 수 있는 자료구조를 탐색할 것이다. 마지막으로 기존에 제안된 다양한 자료구조와 알고

리즘을 실제 IP주소와 하나의 플랫폼에서 시뮬레이션 하여 본 논문에서 제안된 자료구조 및 상호간의 성능을 비교·분석할 것이다.

참 고 문 헌

[1] Peter Newman, Greg Minshall, and Larry Huston, "IP Switching and Gigabit Routers," *IEEE Communications Magazine*, January 1997

[2] Incrnet2, http://www.internet2.edu

[3] www.intel.com/design/PentiumII/

[4] Tong-Bi Pei and Charles Zukowski, "Putting Routing Tables in Sillicon," *IEEE network Magazine*, January 1992

[5] A.J. McAuley and P. Francis, "Fast routing table lookup using CAMs," *In Proceedings of IEEE Infocom'93*, v3, 1382-1391, San Francisco, 1993

[6] David C. Feldmeier, "Improving gateway performance with a routing-table cache," *In proceedings of IEEE Infocom'98*, New Orleans, Louisiana, March 1998

[7] Anthony J. Bloomfeld NJ McAuley, Paul F. Lake Hopatcong NJ Tsuchiya, and Daniel V. Rockaway Township Morris Country NJ Wilson, "Fast Multilevel hierarchial routing table using content-addressable memory," U.S. Patent serial number 034444

[8] P. Gupta, et al., "Routing Lookups in Hardware at Memory Speeds," *In Proceedings of IEEE Infocom'98*, San Francisco, April 1998

[9] A. Moestedt, et al., "IP Address Lookup in Hardware for High-Speed Routing," *Hot Interconnects*, August 1998

[10] A. Bremler-Barr, Y. Afek, and S. Har-Peled, "Routing with Clue," *In Proceedings of ACM SIGCOMM 99*, Cambridge, September 1999

[11] Donald R. Morrison, "PATRICIA - Practical Algorithm to Retreive Information Coded In Alfanumeric," *journal of the ACM*, 15(4):514-534, October 1968

[12] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen pink, "Small Forwarding Tables for Fast Routing Lookups," *In Proceedings of ACM SIGCOMM'97*, October 1997

[13] B. Lampson, V. Srinivasan and G. Varghese, "IP Lookups using Multiway and Multicolumn Search," *In Proceeding of INFOCOM'98*, March 1998

[14] S. Venkatachary and G. Varghese, "Faster IP Lookups using Controlled Prefix Expansion," *In Proceedings of ACM Sigmetrics'98*, June 1998

[15] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner, "Scalable High Speed IP Routing Lookups," *In Proceedings of ACM SIGCOMM'97*, October 1997

[16] Keith Sklower, "A Tree-Based Routing Table for Berkeley Unix," Technical report, University of California, Berkeley

[17] S. Nilsson and G. karlsson, "Fast Address Look-Up for Internet Routers," *In Proceedings of IEEE Broadband Communications'98*, April 1998

[18] T. Kijkanjanarat and H.J.Chao, "Fast IP Lookups using a Two-Trie Data Structure," *In Proceedings of Globecom'99*, 1999

[19] E. Rosen, et al., "Multiprotocol Label Switching Architecture," ftp://ds.internic.net/internet-drafts/draft-ietf-mpls-arch-07.txt, July 2000

[20] Yakov Rekhter et al, "Tag switching architecture overview," ftp://ds.internic.net/internet-drafts/draft-rfcedinfo-rekhter-00.txt, 1996

[21] Peter Newman, Tom Lyon, and Greg Minshall, "Flow labeled IP: a connectionless approach to ATM," *In Proceedings of IEEE Infocom'96*, San Francisco, California, March 1996

[22] Pinar A. Yilmaz, Andrey Belenkiy, Necdet Uzun, and Nitin Gogate, A Trie-based Algorithm for IP Lookup Problem , *In Proceedings of Globecom'00*, 2000

[23] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, "Data Structure and Algorithms," *Addison-Wesley*, 1983

[24] K. Mehlhorn, S. Naher, and H.Alt, "A lower bound on the complexity of the union-split-find probelm," *SIAM Journal on Computing*, December 1988

[25] Michigan University and merit Network, Internet Performance Management and Analysis (IPMA) project, http://nic.merit.edu/~ipma

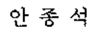
[26] Stanford University Workshop on Fast Routing and Switching, December 1996, http://tinytera.stanford.edu/Workshop_Dec96/

[27] Y. Rekhter and T. Li. "A Border Gateway Protocol 4 (BGP-4)," *IETF RFC 1771*, March 1995

[28] S. Deering, and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," *IETF RFC 2460*, December 1998

오 승 현

1988년 2월 동국대학교 전자계산학과(학사). 1998년 2월 동국대학교 컴퓨터공학과(석사). 1998년 ~ 현재 동국대학교 컴퓨터공학과 박사과정. 1988년 ~ 1996년 (주)대우엔지니어링 시스템부. 관심분야는 실시간 프로토콜, 네트워크 시뮬레이션, IPv6, 무선통신



안 종 석

1983년 서울 공대 전자공학과(학사). 1985년 한국과학기술원 전기 및 전자과(석사). 1995년 University of Southern California 컴퓨터공학과(박사). 1983 ~ 1995년 삼성전자 선임연구원. 1996년 ~ 현재 동국대학교 정보산업학부 조교수. 관심분야는 실시간 프로토콜, 네트워크 사뮬레이션, 멀티케스트, 무선통신