복수의 작은 트리에 대한 바이너리 검색을 이용한 IP 주소 검색 구조

준회원 이 보 미, 정회원 임 혜 숙*, 준회원 김 원 정

Binary Search on Multiple Small Trees for IP Address Lookup

Bomi Lee *Associate Members*, Hyesook Lim* *Regular Members*,
Wonjung Kim *Associate Members*

요 약

인터넷 접근 기술의 발달은 보다 큰 인터넷 bandwidth를 필요로 하게 되었고 라우터에서의 빠른 패킷 처리를 요구하게 되었다. 라우터에서의 어드레스 검색은 일초에 수천만개 이상으로 입력되는 패킷에 대하여 실시간으로 처리되어야 하는 중요한 기능이다. 본 논문에서는 효율적인 어드레스 검색을 위한 소프트웨어에 기반한 새로운 검색 구조를 제안한다. 라우팅 테이블을 여러 개의 밸런스 트리들로 구성하여 하나의 SRAM에 저장하고 각각의 트리에 대해 순차적으로 바이너리 검색을 수행하는 방식이다. 본 논문에서 제안하는 구조는 약 40,000 여개의 프리픽스를 갖는 라우팅 테이블을 저장하기 위하여 301.7KByte의 작은 사이즈 SRAM을 사용하고 평균 11.3번의 메모리 접근으로 주소 검색이 가능한 구조이다.

Key Words : IP Address Lookup, Binary Search, Multiple Balanced Trees, Enclosure Prefix, Disjoint Prefix

ABSTRACT

Advance of internet access technology requires more internet bandwidth and high-speed packet processing. IP address lookups in routers are essential elements which should be performed in real time for packets arriving tens-of-million packets per second. In this paper, we proposed a new architecture for efficient IP address lookup. The proposed scheme produces multiple balanced trees stored into a single SRAM. The proposed scheme performs sequential binary searches on multiple trees. Performance evaluation results show that proposed architecture requires 301.7KByte SRAM to store about 40,000 prefix samples, and an address lookup is achieved by 11.3 memory accesses in average.

Ⅰ. Introduction

Cable television이나, DSL, optical fiber와 같은 인터넷 접근 기술의 발달은 보다 큰 인터넷 bandwidth를 필요로 하게 되었고 라우터에서의 빠른 패킷 처리를 요구하게 되었다[1]. 어드레스 검색은 들어온 패킷의 목적지 IP 어드레스에 따르는 출력 포트를 결정하는 동작으로 라우터의 패킷처리 속도를 결정하는 주요 기능이다. IP 주소의 네트워크 부분을 프리픽스라고 하는데, 기존의 classful addressing 방식은 프리픽스 길이를 8, 16, 혹은 24 로 고정시켜 사용하는 방식으로 기존의 Layer2를 위한 exact match 검색 방식

* 이화여자대학교 정보통신학과 SoC Design 연구실 (hlim@ewha.ac.kr)
논문번호 : #030348-0811, 접수일자 : 2003년 6월 11일
※ This research was partially supported by the MIC(Ministry of Information and Communication), Korea, under the Chung-Ang University HNRC-ITRC(Home Network Research Center) support program supervised by the IITA(Institute of Information Technology Assessment).

을 쉽게 적용할 수 있었다. 반면 classful addressing 방식은 프리픽스 길이에 융통성을 가질 수 없어, IP 주소가 낭비되고 라우트 테이블의 크기가 증가하는 단점을 갖는다. 이를 해결하기 위해 프리픽스 길이를 고정시키지 않은 CIDR (Classless Interdomain Routing) 방식이 등장하게 되었다. CIDR 에서는 네트웍에 들어가는 호스트의 수에 따라 프리픽스 길이를 정할 수 있어, IP 주소의 낭비를 막고, 여러 개의 작은 네트웍을 aggregate 할 수 있어 라우팅 테이블의 크기가 증가하는 것을 방지하는 장점이 있다. 그러나 입력 패킷이 프리픽스 길이에 대한 정보를 가지고 있지 않으므로 가장 specific한 네트워크를 찾기 위해 longest prefix match (LPM)를 수행하여야 하고, 이를 위한 효과적인 알고리듬이 필요하게 되었다. IP 어드레스 검색을 위한 알고리듬은 어드레스 검색을 위한 메모리 접근 횟수와 라우팅 테이블을 저장하기 위한 메모리 크기, 테이블 업데이트, 그리고 pre-processing 필요 여부 등의 여러 요소들을 만족시켜야 한다.

본 논문에서는 효율적인 어드레스 검색을 위한 소프트웨어에 기반을 둔 새로운 검색 구조를 제안한다. 라우팅 테이블을 여러 개의 밸런스 트리들로 구성하여 하나의 SRAM에 저장하고 각각의 트리에 대해 바이너리 검색을 수행하는 방식이다. 본 논문에서 제안하는 구조는 약 40,000 여개의 프리픽스를 갖는 라우팅 테이블을 저장하기 위하여 301.7KByte의 작은 사이즈 SRAM을 사용하고 평균 11.3번의 메모리 접근으로 주소 검색이 가능하며, pre-processing을 요구하지 않아, 라우팅 테이블의 incremental update가 가능한 구조이다. 본 논문의 구성은 다음과 같다. 2장에서는 기존의 연구 방식들에 대해 살펴보고 3장에서는 본 논문이 제안하는 구조에 대해 살펴본다. 4장에서는 기존의 소프트웨어에 기반한 방식들과의 성능을 비교해 본 후 5장에서 결론을 맺는다.

Ⅱ. Previous Works

주소 검색을 위한 연구들은 몇가지 범주로 구분될 수 있다. 첫째는 활발히 연구되고 있는 방식으로 Trie 구조에 기반한 것이다. Trie 구조는 그림1(a)와 같은 모습으로 표현되며 프리픽스를 Trie의 노드에 저장하는 방식이다. 그림1의 각 트리는 그림 아래의 동일한 샘플 프리픽스를 이용하여 구성한 것이다. Trie 는 간단한 구조를 갖지만 중간에 비어있는 노드들이 존재하며, W를 트리의 깊이라 할 때, 메모리 검색 횟수가 O(W)가 필요한 단점이 있다[2]. 그림1(b)의 "Path compressed trie"는 비어있는 노드이면서, 하나의 child를 갖는 노드를 압축시켜 최대 메모리 접근 횟수를 줄이고자 하였다. 그림1(c)의 "Multibit trie"는 메모리 접근 횟수를 줄이기 위해 하나의 노드가 가질 수 있는 child의 갯수를 늘리는 방법이다. 이 방식의 경우 K-bit Trie 라고 할 때 메모리 접근 횟수는 O(W/K)로 감소하나 Trie 의 저장 방법이 복잡해지는 단점이 있다[3]. Trie 를 변형한 다른 방식으로는 위의 두 가지 방법을 모두 적용시킨 "Level Compressed Trie (LC-trie)"가 있다. 그림1(d)의 LC-Trie의 경우 기존의 트리에 비해 좀 더 compact한 구성이 가능해졌으나 여전히 최대 O(W/K)의 메모리 접근이 필요하다[4]. SFT[5]는 4만여개의 라우트 정보를 150-160Kbyte의 캐쉬 사이즈에 저장할 수 있는 새로운 포워딩 테이블 구조를 제안하고 있다. 높이가 32인 Binary Trie를 높이 16에서 잘라 bit mask를 지정한 후 bit mask의 offset을 저장하는 방식으로 컴팩트한 데이터 구조와 빠른 검색을 가능하게 하였다. 그러나 캐쉬의 특성상 큰 라우팅 테이블에 적합하지 않고 복잡한 pre-processing이 요구되어 incremental update가 불가능하다.

IP 어드레스 검색을 위한 두번째 범주는 해슁에 기반한 방식이다. 해슁은 exact matching을 수행하는 layer 2에서 사용되어 왔다. Waldvogel[6]은 바이너리 검색과 해슁을 결합시킨 방식을 제안하였다. 프리픽스 데이터들을 프리픽스 길이 별로 정리한 후, 프리픽스 길이에 대해서는 바이너리 검색을, 프리픽스 값에 대해서는 해슁을 적용하는 방식이다.

이 방식은 길이가 다른 프리픽스들간의 온전한 바이너리 검색이 불가능하여 추가적인 정보를 나타내기 위한 마커와 현재 엔트리의 best matching prefix (BMP)를 저장해야하며 이를 계산하기 위한 pre-processing을 요구한다. 또한 프리픽스 분포에 따른 perfect 해쉬 함수를 빠른 시간에 찾아냄을 가정하고 있으므로 실용적이지 않다. Lim[7]은 프리픽스 길이별로 별도의 테이블과 해슁함수를 사용하여 모든 프리픽스 길이에 대한 병렬 검색을 수행하는 방법을 제안하고 있다. Perfect 해쉬 함수를 가정하지 않고 충돌이 발생한 경우를 고려한 실용적인 구조를 제안하였으나 충돌이 발생한 경우 전체 메모리 검색 횟수가 증가하는 단점을 가지고 있다. Yu[8]은 프리픽스 분포 특성과 해슁을 이용한 방식을 제안한다. 라우팅 테이블은 길이가 24인 프리픽스가 많은 비율을 차지하므로 네 개의 메모리에 나눠 저장하며 나머지 길이

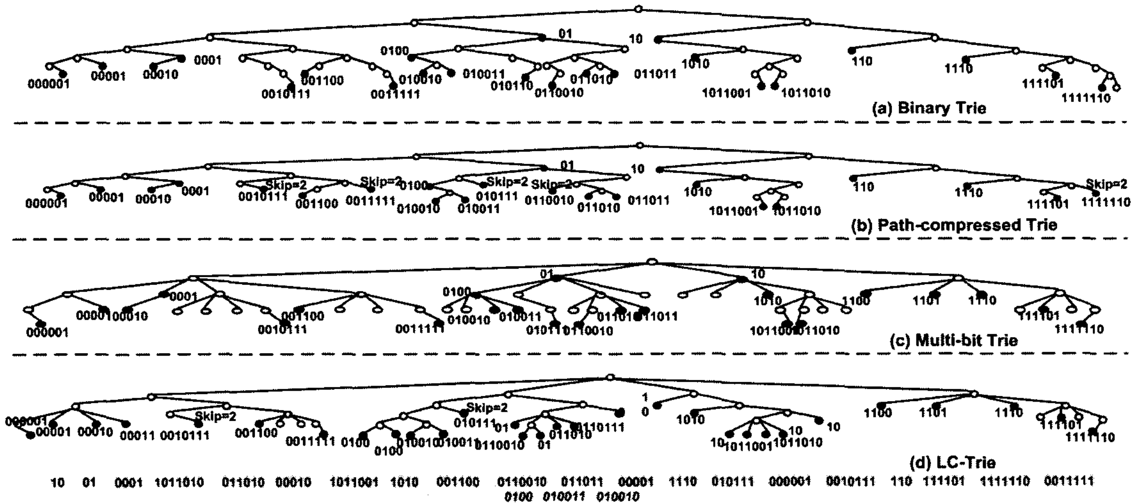


그림 1. 여러가지 Trie 구조

의 프리픽스도 네개의 메모리에 나눠 저장한다. 이 방식은 네 개 메모리를 순차적으로 검색하였을 때 평균 메모리 접근 횟수는 25.6번이며 1500KB의 큰 사이즈의 메모리를 필요로 하는 단점이 있다. Lim[9]은 [7]와 마찬가지로 프리픽스 길이별로 별도의 테이블을 사용하고 모든 길이의 프리픽스에 대해 각각 복수개의 해쉬 인덱스 값을 찾아내는 방법을 제안하고 있다. 복수개의 해쉬 인덱스에 따르는 복수개의 테이블을 사용하므로써 하나의 테이블을 사용하였을 때 보다 충돌이 줄어드는 장점을 가지고 있다. [9]에서 제안한 방식은 하드웨어로 구현되기에 적합한 여러 개의 작은 사이즈의 메모리를 사용하며 평균 1번의 메모리 접근으로 어드레스 검색이 가능한 구조이나 각각의 메모리 사이즈가 프리픽스 분포에 영향을 받으므로 deterministic 하지 않은 단점이 있다.

어드레스 검색을 위한 세번째 범주로는 하드웨어에 기반한 방식이 있다. 하드웨어에 기반한 방식의 경우 1-2번의 메모리 접근으로 검색이 가능하나 특별히 제작된 하드웨어를 필요로 하여 비용이 비싸고 scalability가 떨어지는 단점이 있다. Gupta[10]은 프리픽스 길이가 24와 같거나 짧은 프리픽스를 2^{24} 의 엔트리를 갖는 하나의 메모리에, 길이가 24보다 긴 프리픽스를 다른 하나의 메모리에 저장하는 방식을 제안하고 있다. 프리픽스의 분포가 24에서 높은 특성을 이용한 방식으로 최대 두 번의 메모리접근으로 주소 검색이 가능하나 33MB의 큰 메모리를 필요로 하며 IPv6로의 전환이 어렵다. Huang[11]은 2^{16}개의 인덱스를 가지는 메모리를 사용하여 프리픽스를 저장한 후 16bit보다 긴 프리픽스들에 대해 각각의 서브트리를 구성하는 방식을 제안하였다. 이 방식 역시 최대 3번의 메모리 접근으로 검색이 가능하나 450KB의 메모리를 요구하며 IPv6로의 전환이 어렵다고 할 것이다.

마지막으로 새로운 개념의 트리 구조에 기반한 방식이다. Yazdani[12]는 길이가 다른 프리픽스의 바이너리 검색을 가능하게 하는 그림2와 같은 정의를 제안하였다.

[12]에서는 그림2의 정의를 이용하여 그림3(a)의 트리구조를 제안하였다. 그림3(a)의 트리 역시 그림1과 동일한 샘플 프리픽스를 사용하여 구성하였다. 그림 3(a)에서 볼 수 있듯이 [12]의 트리는 binary trie(그림1)와 달리 비어있는 노드가 없어 메모리를 효율적으로 사용할 수 있는 장점을 지닌다. 그러나 인클로저와 인클로저의 서브-프리픽스들이 하나의 트리에 포함되기 때문에 언밸런스한 구조의 트리가 형성되어 트리의 깊이가 깊어지며, W를 트리의 깊이라 할 때, 최대 O(W)번의 메모리 검색이 필요한 단점이 있다. 또한 단순 binary 검색을 적용할 수 없기 때문에 각 노드는 자신의 children으로 가는 포인터들을 기억해

정의 1 Compare
a. 두 프리픽스의 길이가 같으면 numerical 값이 비교된다.
Ex) A=1001 B=1100인 경우 B>A
b. 두 프리픽스의 길이가 다른 경우 짧은 길이까지만 비교된다.
두 프리픽스의 substring 값이 같은 경우, 다음 bit이 1이면 B>A, 아니면 B<A 이다.
Ex) A=1001, B=110000인 경우 B>A
Ex) A=1001, B=100110인 경우 B>A
Ex) A=1001, B=100100인 경우 A>B
정의 2 Match
두 프리픽스가 같거나 짧은 프리픽스 길이까지가 같은 경우 매치한다.
Ex) A=1001, B=100100인 경우 A와 B는 매치한다.
정의 3 Disjoint
두 프리픽스가 매치하지 않으면 A와 B는 disjoint 하다.
Ex) A=1001, B=111인 경우 A와 B는 disjoint이다.
정의 4 Enclosure
프리픽스 A를 프리픽스로 갖는 다른 프리픽스가 하나라도 존재하면 A는 enclosure이다.
Ex) A=1001, B=100100인 경우 A는 B의 enclosure이다.

그림 2. Yazdani[12]의 정의

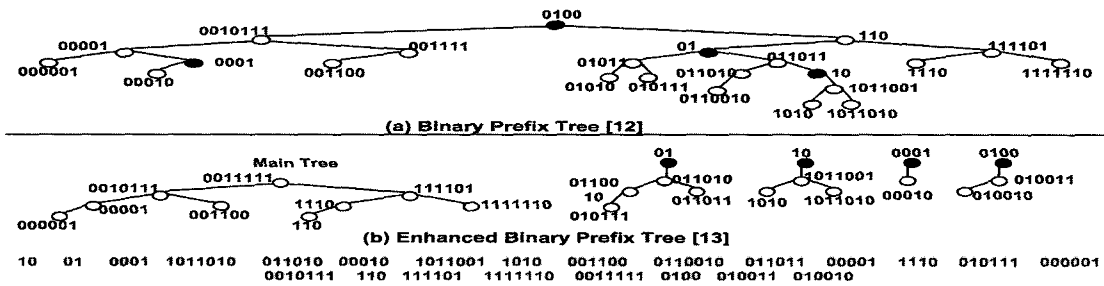


그림 3. BPT[12]와 Enhanced Binary Tree[13] 구조

야 하는 단점이 있다. 그림3(b)는 Lim[13]에서 제안하는 EnBiT (Enhanced Binary Tree) 트리를 보여준다. Lim[13]은 디스조인트 프리픽스들은 밸런스된 트리를 구성할 수 있다는 점에 착안하여 여러 개의 밸런스 트리를 사용한 방식을 제안하고 있다. [13]에서 제안하는 구조는 트리구조에 파이프라인을 적용시켜 한번의 메모리 접근으로 주소 검색이 가능한 장점을 가지고 있으나 각 레벨의 SRAM 사이즈가 라우팅 테이블의 프리픽스 분포 특성에 따라 결정되는 단점을 가지고 있다.

본 논문에서는 디스조인트 프리픽스들로 구성된 트리는 밸런스 트리를 구성하여 온전한 바이너리 검색이 가능하다는 점에서 착안하여, 여러 개의 밸런스 트리들을 하나의 메모리에 저장하여 각 트리마다 순차적인 바이너리 검색을 적용하는 방식을 제안한다. 제안하는 구조는 앞서 설명한 [12]와 [13]의 단점을 모두 해결한 구조로서, 밸런스 트리의 각 노드들에 노드의 children으로 가는 포인터를 저장할 없이 메모리의 인덱스를 사용하여 바이너리 검색이 수행되므로 메모리 사용 효율이 매우 증가한다. 그러므로 캐쉬에 저장 될 수 있는 충분히 작은 사이즈의 메모리를 필요로 하고, 라우팅 테이블을 위한 메모리 크기가 프리픽스의 길이가 아닌 개수에 따라 결정되어 IPv6로의 전환이 용이하다. 검색 횟수도 N을 프리픽스의 갯수라 할 때, $O(\log_k N)$인 특징 (바이너리 검색인 경우 $k = 2$)을 갖는데, Range Table을 사용하여 검색 범위를 줄인 경우 검색 횟수를 보다 줄일 수 있다. 제안하는 구조는 소프트웨어에 기반을 둔 방식으로 일반 프로세서로 처리 가능한 flexible한 구조이다.

Ⅲ. The Proposed Scheme

1. Prefix 분류

라우팅 테이블의 프리픽스는 앞서 설명한 인클로저와 디스조인트 정의를 이용하여 분류된다. 그림4는 그림3의 프리픽스들을 분류한 것으로서, 분류 결과 각 프리픽스에 대한 레벨과 type 정보를 얻을 수 있다. 프리픽스의 레벨은 프리픽스가 인클로저의 서브트리에 포함될 때 증가한다. 예를 들어 그림4의 01*은 첫번째 레벨 인클로저이고 0100*은 두번째 레벨 인클로저가 되며, 010011*는 세번째 레벨 디스조인트 프리픽스가 된다. 첫번째 레벨 인클로저와 첫번째 레벨 디스조인트 프리픽스는 메인트리를 구성하고, 두번째 레벨 디스조인트 프리픽스와 두번째 레벨 인클로저 프리픽스들은 첫번째 레벨 인클로저의 서브트리가 된다.

2. Building scheme

앞서 설명한 방식으로 분류된 프리픽스는 그림5의 Main Table에 저장된다. 프리픽스 중 첫번째 레벨의 인클로저 프리픽스와 디스조인트 프리픽스를 바이너리 검색이 가능하도록 크기순서대로 Main Table의 첫 부분에 저장한다. Main Table에 저장된 인클로저들은 각각의 서브트리를 가리키는 포인터와 서브 프리픽스의 개수에 대한 정보를 가지고 있다. 인클로저의 서브 트리 역시 크기 순서대로 메모리에 저장되고 서브프리픽스 중 인클로저 프리픽스가 있는 경우 그 프리픽스는 다시 자신의 서브트리를 가리키는 포인터와 개수에 대한 정보를 갖는다. 디스조인트 프리픽스의 경우에는 서브트리에 대한 정보를 갖지 않는다. 그림 5의 Range Table은 메인 테이블에 저장되는 프

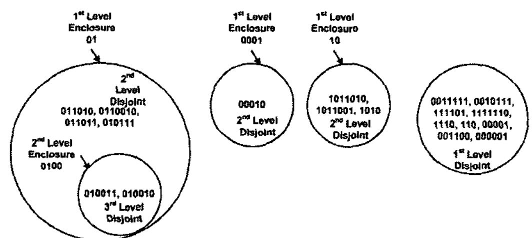


그림 4. 프리픽스 분류

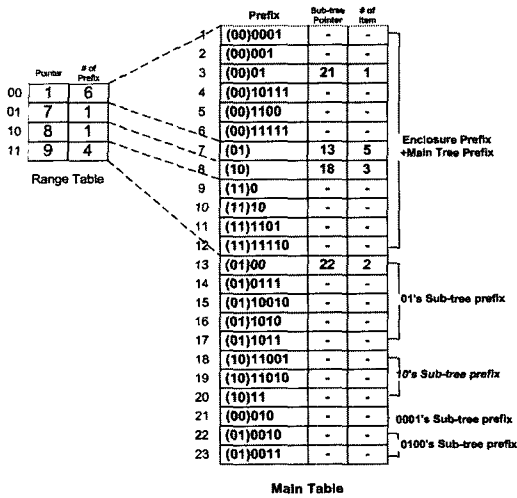
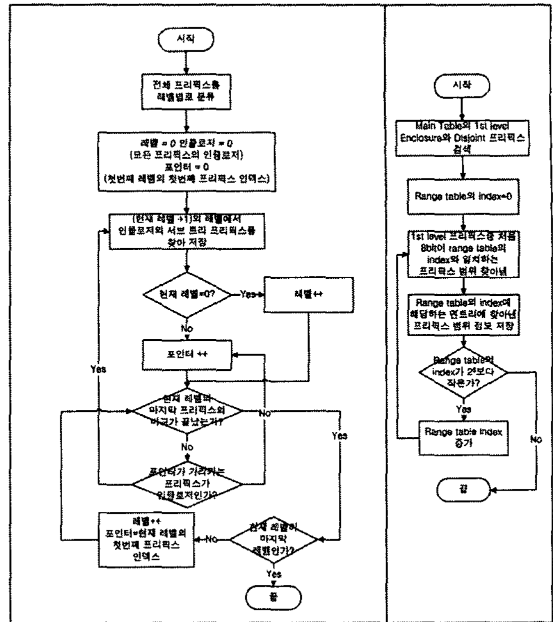


그림 5. 제안하는 구조의 프리픽스 저장 방식

리픽스들의 처음 2 비트에 대한 정보를 담고 있는데, 처음 2 비트가 Range Table의 메모리 인덱스에 해당하는 프리픽스의 갯수와 위치를 지정한다. 프리픽스의 처음 2비트는 Range Table의 인덱스로 사용되므로, Main Table에는 처음 2비트를 저장할 필요 없이 프리픽스의 나머지 부분만 저장하면 되며, Main Table의 프리픽스에 괄호를 사용하여 표현하였다. 그림6은 제안하는 구조의 빌드 과정을 보여주는 플로차트이다.

3. Search scheme

32 비트 IP 어드레스 검색을 위하여, 제안하는 구조에서는 IPv4 프리픽스 중 가장 짧은 길이인 8 비트 인덱스를 갖는 Range Table을 사용하였다. 제안하는 구조의 검색은 Range Table에서 시작한다. 입력된 패킷의 destination IP 주소의 처음 8bit을 Range Table의 인덱스로 사용하여 Range Table에서 Main Table의 검색 범위 정보를 얻게 되며, 그 범위에서 첫번째 레벨 인클로저와 디스조인트들 간의 바이너리 검색을 수행하고 매치하는 프리픽스를 찾는다. 매치하는 디스조인트 프리픽스를 찾은 경우 검색이 종료되는데, 첫번째 레벨 바이너리 검색에서 매치하는 디스조인트 프리픽스를 찾은 경우 해당 엔트리의 아웃 포트 포인터를 출력한 후 검색을 끝마치고, 매치하는 인클로저를 찾은 경우 인클로저의 아웃 포트 포인터를 저장한 후, 인클로저 서브트리로 이동하여 다시 바이너리 검색을 수행하게 된다. 인클로저의 서브트리에서 매치하는 엔트리가 있는 경우 아웃 포트 포인터 값을 업데이트 하게 된다. 그림5의 예로 살펴보면 입력된 주소



(a) Main Table Build (b) Range Table Build

그림 6. 제안하는 구조의 빌드 플로차트

가 0001010인 경우 입력 주소의 처음 2bit이 00이므로 Range Table로부터 Main Table의 메모리 주소 1부터 6개의 프리픽스에 대한 검색을 수행해야 함을 알 수 있다. 주소 1부터 6개의 첫번째 레벨의 인클로저와 디스조인트에 대해 binary 검색을 수행한 결과 인클로저 (00)01*과 매치하므로 (00)01*의 서브트리 포인터가 가리키는 서브트리로 이동하여 계속 검색을 수행한다. (00)01*의 서브트리 검색 결과 두번째 레벨 디스조인트인 (00)010*과 매치하므로 00010*은 입력된 어드레스의 LPM이 되며, 00010*의 아웃 포트 포인터를 출력한 후 검색이 종료된다. 그림7은 제안하는 구조의 검색 플로차트이다.

4. Update

제안하는 구조는 업데이트를 위해 각각의 서브트리 사이에 일정 비율의 엔트리를 비워 놓음을 가정하였다. 또한 프리픽스의 처음 8bit은 Main Table에 저장하지 않으며, Main Table 저장과 함께 Range Table의 정보를 업데이트 시킨다. 예를 들어 0010* 프리픽스를 추가하고자 하는 경우 먼저 Range Table을 통해 00으로 시작하는 프리픽스의 범위를 알아낸다. 0010*은 범위 안의 첫번째 레벨 인클로저, 디스조인트 프리픽스와 모두 디스조인트 관계이므로 크기순서에 맞춰 저장된다. 또한 00으로 시작하는 프리픽스가 추가되었으므로 Range Table의 정보를 update 시킨

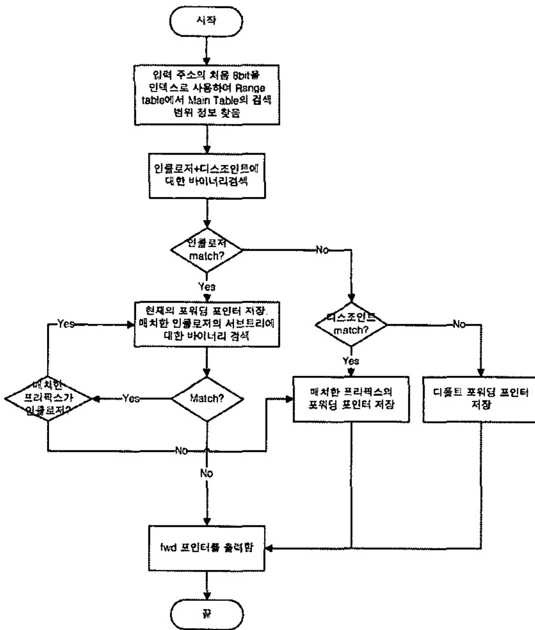


그림 7. 제안하는 구조의 검색 플로차트

다. 100*프리픽스를 추가하고자 하는 경우 100*은 10*의 서브 프리픽스이므로 10*의 서브트리로 이동한다. 10*의 서브 프리픽스들과 100*은 디스조인트 관

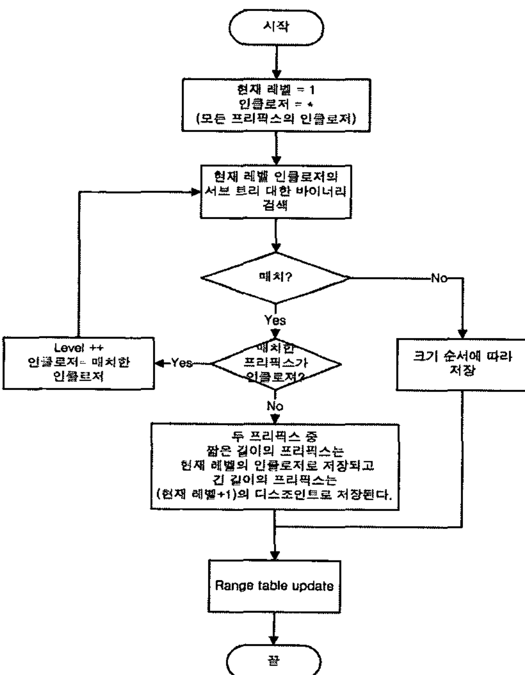


그림 8. 제안하는 구조의 업데이트 플로차트

계이므로 100*은 서브트리에 크기 순서에 맞게 저장하고 Main Table의 정보를 update시킨다. 1100*을 삽입하는 경우 1100*은 첫번째 레벨 디스조인트 프리픽스인 110*의 서브프리픽스가 되므로 110*은 인클로저가 되고 1100*을 서브트리로 기억하게 된다. 1111*을 추가하는 경우 1111*은 첫번째 레벨 디스조인트인 111101*과 1111110*을 서브프리픽스로 갖는 인클로저가 되므로 첫번째 레벨 인클로저로 저장되고 111101*과 1111110*은 두번째 레벨 디스조인트로 다시 저장된다. 그림8은 업데이트 동작을 나타낸 플로차트이다.

5. 엔트리 구조

제안하는 방식의 엔트리 구조는 그림 9와 같다. Main Table의 경우 인클로저 프리픽스의 서브트리를 가리키기 위한 서브트리 포인터와 바이너리 검색을 위한 서브트리의 프리픽스 개수를 나타내는 값을 가지고 있다. Range Table은 해당 범위의 프리픽스가 시작하는 주소와 범위의 프리픽스 개수에 대한 정보를 갖고 있다.

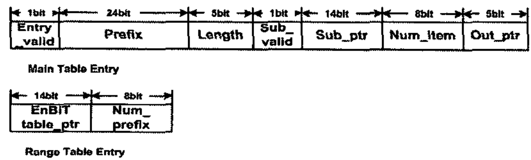


그림 9. 엔트리 구조

6. Extension to multiway search

본 논문에서 제안하는 구조는 하나의 엔트리에 여러 개의 프리픽스를 저장하는 방식을 적용하여 multiway 검색으로 확장할 수 있다. 각 엔트리에 하나의 프리픽스를 저장한 바이너리 검색이 한번의 메모리 접근으로 전체 검색범위를 1/2로 줄인다면, 엔트리에 K-1개의 프리픽스를 저장한 multiway 검색의 경우 한번의 메모리 접근으로 전체 검색범위를 1/K로 줄이는 특징이 있다. 즉 하나의 엔트리에 K-1개의 프리픽스를 저장한 경우, N개의 prefix를 위해, 전체 트리의 깊이를 $O(\log_K N)$로 줄일 수 있어 검색 횟수가 줄어드는 장점이 생긴다. IPv4 프리픽스 검색을 위해서는 multiway 를 적용하고, IPv6를 위해서는 바이너리 검색을 적용한다면, 라우팅 엔트리의 폭을 같게 하여 하나의 메모리에 IPv4와 IPv6 라우팅 테이블을 구성할 수 있을 뿐 아니라, 많은 라우팅 데이터를 갖

표 1. 라우팅 테이블 분석

	Routing Entries	1st level Enclosure/ Disjoint Prefix	2nd level Enclosure/ Disjoint Prefix	3rd level Enclosure/ Disjoint Prefix	4th level Enclosure/ Disjoint Prefix	5th level Enclosure/ Disjoint Prefix	Total Enclosure / Disjoint Prefix
MAE-WEST (1)	14553	260 / 12863	5 / 1420	0 / 5	-	-	265 / 14288
MAE-WEST (2)	14937	315 / 13083	5 / 1526	0 / 8	-	-	320 / 14617
AADS	20204	371 / 17734	1 / 2097	0 / 1	-	-	372 / 19832
PAC BELL	20519	360 / 18152	2 / 2003	0 / 2	-	-	362 / 20157
MAE-WEST (3)	29584	1599 / 19001	312 / 7538	50 / 941	1 / 140	0 / 2	1962 / 27622
MAE- EAST (1)	37993	1168 / 29244	31 / 7446	0 / 104	-	-	1199 / 36794
MAE-EAST (2)	39464	1120 / 30838	28 / 7427	0 / 51	-	-	1148 / 38316
FUNET	40905	1288 / 24417	297 / 14034	1 2 / 835	0 / 22	-	1597 / 39308

는 IPv4의 어드레스 검색 시간을 줄이는 데 매우 효과적이라 할 것이다.

Ⅳ. Comparison

본 논문에서는 C언어를 사용하여 제안하는 구조의 성능을 실험하였다. 표1은 성능 실험을 위해 사용한 라우팅 테이블 데이터의 관계를분석한 것이다. 표1에 따르면 각 라우터는 비슷한 비율로 디스조인트 프리픽스와 인클로저 프리픽스를 가짐을 알 수 있다.

그림10과 그림 11은 [12]에서 제안한 'Binary prefix tree'와 본 논문에서 제안한 두 가지 방식, Range Table을 사용하지 않은 트리와 Range Table을 사용한 트리의 성능을 비교한 그래프이다. 그림10과 11, 표2에서 볼 수 있듯이 [12]에서 제안한 구조는 4만여개의 라우팅 데이터에 대하여 평균 16.5 번의 메모리 검색을 필요로 하는 반면 range table을 사용한 제안하는 트리의 경우 최대 24번, 평균 11.3번으로 [12]보다 작은 사이즈의 메모리를 사용하여 작은 횟수의 메모리 검색을 필요로 하는 것을 알 수 있다. 실험 결과를 통해 알 수 있듯이 하나의 커다란 트리를 여러 개의 서브 트리로 나누고 또한 Range Table을 이용하여 검색 범위를 줄인 결과, 전체 검색 횟수를 줄일 수 있는 것을 알 수 있다.

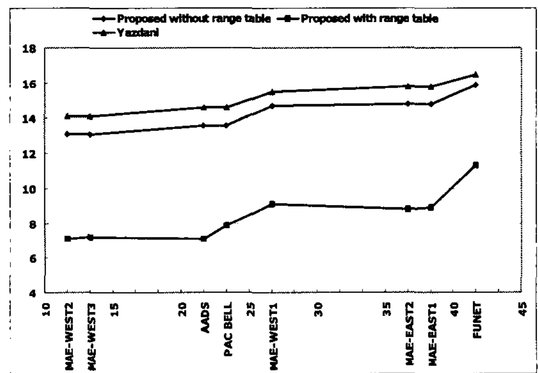


그림 11. [12]와 제안하는 구조의 메모리 접근 횟수 비교 그래프

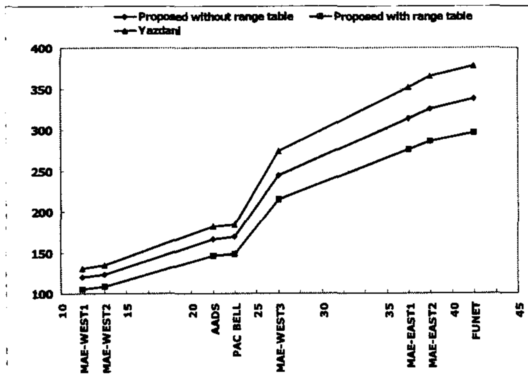


그림 10. [12]와 제안하는 구조의 메모리 사이즈 비교 그래프

그림 12는 [15]에서 40,000여개의 MAE-EAST 라

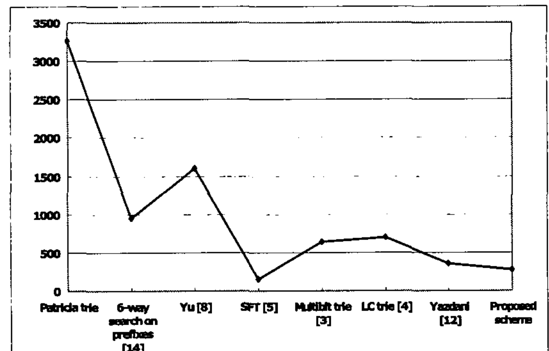


그림 12. 소프트웨어에 기반한 방식들의 메모리 사이즈 비교

표 2. [12]와 제안하는 구조의 메모리 접근 횟수

	Yazdani[12]			Proposed Scheme without range table			Proposed Scheme with range table		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
MAE-WEST (1)	14.1	13	23	13.1	1	21	7.2	1	16
MAE-WEST (2)	14.1	13	22	13.1	1	20	7.1	1	15
AADS	14.6	14	22	13.6	1	22	7.9	1	18
PAC BELL	14.6	14	22	13.6	1	22	8.0	1	18
MAE-WEST (3)	15.5	14	31	14.6	1	28	9.1	1	21
MAE-EAST (1)	15.8	14	25	14.8	1	27	8.8	1	24
MAE-EAST (2)	15.8	14	26	14.8	1	24	9.0	1	21
FUNET	16.5	14	31	15.8	1	29	11.4	1	24

우팅 테이블을 사용한 소프트웨어에 기반한 구조들의 메모리 사이즈와 본 논문에서 제안한 구조의 메모리 사이즈를 비교한 그래프이다. 소프트웨어에 기반한 방식 중 SFT[5] 방식이 160Kbytes로 가장 작은 사이즈의 메모리를 사용하며 본 논문에서 제안하는 구조가 두번째로 작은 사이즈의 메모리를 사용하는 것을 볼 수 있다.

또한 제안하는 구조는 하드웨어에 기반한 방식에 비해 작은 사이즈의 메모리를 사용한다. 하드웨어에 기반한 방식의 경우 33MB와 같이 큰 사이즈의 메모리를 필요로 하게 되고 이 경우 메모리를 chip에 내장할 수 없는 문제점이 생긴다. Chip 외부의 메모리를 사용하는 경우 chip의 pin 수가 늘어나며 메모리 접근 속도가 떨어지는 단점이 있다.

표3은 기존에 제안된 소프트웨어에 기반한 방식들과의 성능을 비교한 결과이다. SFT[5] 방식의 경우 캐쉬 사이즈에 맞는 작은 사이즈의 메모리를사용하지만 최대 검색 횟수가 많이 필요한 것을 볼 수 있다. Yu[8]의 경우 네개의 메모리에 대해 병렬 검색을 수행한 경우 평균 검색횟수는 6.4번이나 본 논문에서 제안하는 방식처럼 하나의 메모리에 저장한 경우 병렬 검색이 불가능 하여 평균 25.6 번의 메모리 검색이 요구되며, 큰 사이즈 메모리를 필요 로 함을 알 수 있다. 본 논문에서 제안한 구조의 경우 다른 소프트웨어 기반 방식들과 비교해 보았을 때 작은 사이즈 메모리를 사용하며 작은 수의 메모리 접근으로 라우팅 검색이 가능한 것을 볼 수 있다. 제안하는 구조는 301.7Kbyte의 작은 사이즈 메모리를 필요로 하며 이는 캐쉬의 최대 사이즈인 512Kbyte보다 작은 사이즈로 캐쉬에 저장되어질 수 있다. Range Table의 경우 0.7Kbyte로 L1 캐쉬에 저장 가능한 크기이므로 전체 메모리 사이즈에서 제외되었다[13].

V. Conclusion

본 논문에서는 디스조인트 프리픽스들 사이에는 바이너리 검색이 가능한 점을 이용하여, 커다란 라우팅 테이블을 디스조인트 프리픽스들로만 이루어진 여러

표 3. Software에 기반한 방식 비교

	# of memory access			Storage	Pre-processing	Incremental Update
	Avg	Min	Max			
SFT[5]	Not Available	5	40	160KB	Required	Not Possible
Yu[8]	25.6	8	104	1500KB	Not Required	Possible
Yazdani[12]	16.4	14	31	378.4KB	Not Required	Not Possible
Proposed Scheme	11.3	1	24	301.7KB	Not Required	Possible

개의 트리로 만들어, 하나의 메모리에 저장하고, 여러 개의 트리에 대한 바이너리 검색을 순차적으로 수행하는 구조를 제안한다. 또한 Range Table을 사용하여 검색 범위를 더욱 줄여 평균 검색 횟수를 줄일 수 있음을 보였다. 제안하는 구조는 소프트웨어에 기반한 방식으로 특정 하드웨어나 프로세서가 필요 없는 일반적인 구조로 낮은 가격에 구현이 가능하다. 본 논문에서 제안된 구조는 프리픽스의 길이와 상관없이 라우팅 테이블의 엔트리 수에 따라 전체 요구되는 메모리 크기 및 메모리 접근 횟수가 결정되므로 IPv6로의 전환이 용이하다. 실제 라우터 데이터를 이용하여 실험한 결과, 약 40,000개의 라우팅 데이타를 저장하기 위하여, 본 논문에서 제안하는 구조는 11번의 평균 메모리 접근과 301.7KByte의 캐쉬에 저장될 수 있는 작은 사이즈의 메모리를 요구함을 보았다

참 고 문 헌

[1] H.Jonathan Chao, "Next Generation Routers", Proceedings of the IEEE, Vol.90, No.9, pp.1518-1558, Sep, 2002

[2] 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

[3] V.Srinivasan and G.Varghese, "Faster IP Lookup using Controlled Prefix Expansion", in Proc. ACM SIGMATICS'98, pp.1-10, June 1998

[4] S.Nilsson and G.Karlsson, "IP-Address Lookup using LC-tries", IEEE Journal on Selected Area in Communication, Vol.17, pp.1083-1092, Jun 1999

[5] M.Degermark, A.Brodnik, S.Carlsson, S.Pink, "Small Forwarding Tables for Fast Routing Lookups", Proc. ACM SIGCOMM, pp.3-14, 1997

[6] M.Waldvogel, G.Varghese, J. Turner, and B.Plattner, "Scalable High Speed IP Routing Lookups", in Proc. ACM SIGCOMM'97 Conf., pp. 25-35, 1997

[7] Hyesook Lim, Ji-Hyun Seo, and Yeo-Jin Jung, "High speed IP Address Lookup Architecture Using Hashing", 한국 통신학회지, 28권, 2B, pp.138-143, 2003

[8] Daxiao Yu, Brandon C. Smith and Belle Wei, "Forwarding Engine for Fast Routing Lookups and Updates", Proc.IEEE GLOBECOM'99, pp.1556-1564, 1999

[9] Hyesook Lim and Yeojin Jung, "A Parallel Multiple Hashing Architecture for IP Address Lookup", Proc. IEEE HPSR2004, pp.91-98, 2004

[10] P.Gupta, S.Lin and N.McKeown, "Routing lookups in hardware at memory access speed", Proc. IEEE INFOCOM'98, pp 1240-1247, ARP 1998

[11] N.Huang, S.Zhao, J.Pan and C.Su, "A Fast IP Routing Lookup Scheme for Gigabit switching routers", Proc.IEEE INFOCOM'99, Mar 1999

[12] N.Yazdani and P.S.Min, "Fast and Scalable Schemes for the IP Address Lookup Problem", Proc. IEEE HPSR2000, pp 83-92, 2000

[13] Hyesook Lim and Bomi Lee, "A New Pipelined Binary Search Architecture for IP Address Lookup", Proc. IEEE HPSR2004, pp.86-90, Apr. 2004

[14] IA-32 Intel Architecture Software Developer's Manual, Intel, 2004

[15] Butler Lampson, Venkatachary Srinivasan and George Varghese, "IP Lookups Using Multiway and Multicolumn Search", IEEE/ACM Transactions on Networking, pp 324-334, Vol.7, No.3, Jun. 1999

이 보 미 (Bomi Lee) 준회원

2003년 2월 : 이화여자대학교 정보통신학과 학사
2003년 3월 ~ 현재 : 이화여자대학교 정보통신학과 석사과정

<관심분야> Router나 switch등의 Network 관련 SoC 설계, TCP/IP관련 하드웨어 설계

김 원 정 (Wonjung Kim) 준회원

2004년 2월 : 이화여자대학교 정보통신학과 학사
2004년 3월 ~ 현재 : 이화여자대학교 정보통신학과 석사과정

<관심분야> Router나 switch등의 Network 관련 SoC 설계, 멀티미디어 네트워크

임 혜 숙 (Hyesook Lim) 정회원

1986년 2월 : 서울대학교 제어계측공학과 학사
1986년 8월 ~ 1989년 2월 : 삼성 휴렛 팩커드 연구원
1991년 2월 : 서울 대학교 제어계측공학과 석사
1996년 12월 The University of Texas at Austin, Electrical and Computer Engineering 박사
1996년 11월 ~ 2000년 7월 : Lucent Technologies Member of Technical Staff
2000년 7월 ~ 2002년 2월 Cisco Systems Hardware Engineer
2002년 3월 ~ 현재 : 이화여자대학교 정보통신학과 조교수

<관심분야> Router나 switch등의 Network 관련 SoC 설계, 통신관련 SoC 설계