# 스위치 포인터를 이용한 균형 이진 IP 주소 검색 구조

(Binary Search Tree with Switch Pointers for IP Address Lookup)

김 형 기 †  임 혜 숙 ††
(Hyeonggee Kim) (Hyesook Lim)

**요 약** 인터넷 라우터에서의 패킷 포워딩은 라우팅 테이블에 저장되어 있는 다양한 길이의 프리픽스들 중, 입력된 패킷의 목적지 주소와 가장 길게 일치하는 프리픽스를 찾아 그 프리픽스가 지정하는 출력 포트로 입력된 패킷을 내 보내주는 일련의 과정을 말한다. 패킷 포워딩 속도의 관건은 IP 주소 검색 성능이라 할 수 있는데, 고속의 IP 주소 검색을 제공하기 위해서는 포워딩 테이블을 저장하기 위한 효율적인 데이타 구조 및 우수한 검색 알고리즘이 필수적이라 할 수 있다. 본 논문에서는 이진 검색 트리를 이용한 주소 검색 알고리즘에 주목한다. 기존에 나와 있는 모든 이진 검색 알고리즘은 균형 검색을 제공하지 못하여 효율적이지 못하고, 프리픽스 영역에 대한 이진 검색 알고리즘은 균형 검색을 제공하나 프리픽스 개수보다 많은 수의 포워딩 엔트리를 저장하여 또한 효율적이지 못하다. 본 논문에서는 효율적인 IP 주소 검색을 위하여 완전 균형 트리 구조를 만들어 이진 검색을 수행하는 알고리즘을 제안하고, 그 성능을 평가하여 기존의 다른 주소 검색 알고리즘과 비교한다. 성능 평가 결과 본 논문에서 제안하는 알고리즘은 메모리 요구량의 증가 없이 검색 속도가 매우 향상됨을 보았다.

**키워드** : IP 주소 검색, 스위치 포인터, 이진 검색 구조, 완전 균형 트리 구조

***Abstract*** Packet forwarding in the Internet routers is to find out the longest prefix that matches the destination address of an input packet and to forward the input packet to the output port designated by the longest matched prefix. The IP address lookup is the key of the packet forwarding, and it is required to have efficient data structures and search algorithms to provide the high-speed lookup performance. In this paper, an efficient IP address lookup algorithm using binary search is investigated. Most of the existing binary search algorithms are not efficient in search performance since they do not provide a balanced search. The proposed binary search algorithm performs perfectly balanced binary search using switch pointers. The performance of the proposed algorithm is evaluated using actual backbone routing data and it is shown that the proposed algorithm provides very good search performance without increasing the memory amount storing the forwarding table. The proposed algorithm also provides very good scalability since it can be easily extended for multi-way search and for large forwarding tables

**Key words** : IP Address Lookup, Switch Pointer, Binary Search Tree, Balanced Tree

· 본 연구는 지식 경제부 및 정보통신 연구 진흥원의 대학 IT연구 센터(홈네트워크 연구 센터) 육성, 지원 사업의 연구 결과로 수행되었습니다.

† 학생회원 : 이화여자대학교 대학원 전자정보통신학과
calmsea@ewhain.net
†† 정 회 원 : 이화여자대학교 전자정보통신학과 교수
hlim@ewha.ac.kr

논문접수 : 2008년 5월 2일
심사완료 : 2008년 10월 15일

Copyright©2009 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.

정보과학회논문지: 정보통신 제36권 제1호(2009.2)

## 1. 서 론

현재 네트워킹 라우터에서의 패킷 처리 속도는 링크 속도의 발달에 미치지 못하여 패킷의 실시간 처리가 이루어 지지 못하고 있는 실정이다. 이러한 패킷 처리 속도를 개선시키기 위해 이진 트라이(binrary trie)[1], 이진 검색 트리(binary search tree)[2], 가중 이진 검색 트리(weighted binary search tree)[3], 우선순위 트라이(priority trie)[4], 레벨 축약 트라이(level-compressed trie)[5] 등의 여러 알고리즘들이 제안되었으나, 불균형 구조라는 한계 때문에 검색 속도를 크게 향상시키지 못

한다. 따라서 본 논문은 이러한 불균형 구조가 프리픽스들 간의 네스팅 관계를 처리해 주기 위해 발생한다는 점에 착안하여 문제를 해결하고 검색 성능을 향상시켰다. 즉, 제안하는 구조는 이진 검색 트리에 기초한 구조로서, 완전 균형 이진 검색 트리를 구성하되, 스위치 포인터를 사용하여 프리픽스들 간의 네스팅 관계를 처리하는 새로운 접근 방식을 시도한다.

본 논문의 구성은 다음과 같다. 먼저 2장에서 기존에 연구되어 온 알고리즘들을 간략하게 설명한다. 3장에서는 제안된 구조를 설명하고, 4장에서 시뮬레이션 결과로서 제안된 알고리즘의 성능을 평가하고, 타 구조와의 성능을 비교한다. 마지막으로 5장에서 결론을 맺는다.

## 2. 기존의 주소 검색 구조

LPM에 의한 주소 검색이란 라우팅 테이블에 저장되어 있는 다양한 길이의 프리픽스 중 들어온 패킷의 목적지 프리픽스와 일치하는 여러 개의 프리픽스들을 찾아내, 그 중 가장 길게 일치하는 프리픽스를 선정하는 것이다. 따라서 주소 검색 알고리즘의 데이타 구조를 설정하는데 있어 고려해야 할 점은 프리픽스들간의 네스팅 관계이다. 이진 트라이나 이진 검색 트리 구조에서는 어떤 프리픽스가 다른 프리픽스와 네스팅 관계에 있을 때, 짧은 프리픽스를 상위 레벨 노드에 저장하고, 긴 프리픽스를 하위 레벨 노드에 저장하는 방법을 사용하여 프리픽스들 간의 네스팅 관계를 처리한다. 이로써 하위 레벨로 검색을 진행하여 검색 범위를 줄여가면서도 패킷의 목적지 프리픽스와 일치하는 모든 프리픽스들이 검색 범위에 포함되게 된다. 그러나 이렇게 네스팅 관계를 처리한 구조들은 모두 불균형 구조이기 때문에 트라이, 혹은 이진 검색 트리의 깊이가 비효율적으로 커지는 한계가 있으며, 이에 따라 IP 주소 검색 속도가 느린 단점이 있다. 따라서 효율적인 IP 주소 검색을 위해서는 트라이, 혹은 이진 검색 트리의 깊이를 최소화시켜야 하며, 완전 균형 구조를 갖게 하는 것이 가장 최적화 된 것이라 할 수 있다.

이진 검색에 기초한 여러 가지 IP 주소 검색 알고리즘을 설명하기 위하여 표 1과 같은 프리픽스 셋의 예를 보였다. IP 주소의 길이는 IPv4의 경우에는 32 비트, IPv6의 경우에는 128 비트이나, 여기서는 설명의 편의를 위하여 7 비트로 가정하였다. 또한 각 프리픽스에 해당하는 출력 포트는 프리픽스의 이름으로 대신하였다. 먼저 그림 1에 표 1의 프리픽스 셋에 대하여 IP 주소 검색을 위한 가장 단순한 구조인 이진 트라이(binary trie)를 보였다[1]. 이진 트라이 구조에서의 각 프리픽스들은 자신의 길이에 해당하는 레벨의 노드에 저장되며,

표 1 프리픽스 셋

| Prefix Name | Prefix |
|---|---|
| P0 | 01* |
| P1 | 1* |
| P2 | 110100* |
| P3 | 110* |
| P4 | 110110* |
| P5 | 1101* |
| P6 | 1111* |

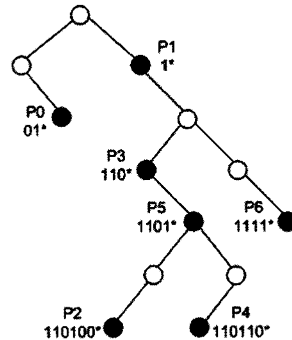


그림 1 이진 트라이의 구조 예시

프리픽스 값에 의하여 트라이의 루트 노드로부터 해당 노드로 가는 패스가 결정됨을 알 수 있다. 그림 1에서 보여준 바와 같이 프리픽스의 분포에 따라 이진 트라이의 구조가 결정되어, 일반적으로 매우 불균형적인 구조가 만들어진다.

이진 트라이 구조는 그림 1에서 볼 수 있듯이 프리픽스를 저장하지 않는 매우 많은 수의 빈 노드를 갖게 되는 단점이 있는데, 우선순위 트라이 구조에서는 모든 비어있는 노드마다, 이 노드를 루트로 하는 트라이에 속한 프리픽스 중 가장 긴 프리픽스를 이동시켜 저장하는 방법으로 트라이의 모든 빈 노드를 제거하였다[4]. 예를 들어 그림 1의 루트 노드는 비어 있는 노드이며, 자신을 루트로 하는 트라이에 속한 프리픽스 중 가장 긴 프리픽스는 P2 혹은 P4이므로 이 두개의 프리픽스 중 임의로 한 개의 프리픽스를 이동시켜 저장하고, 루트의 왼쪽 차일드 노드에는 프리픽스 P0를 이동시켜 저장한다. 이러한 과정을 반복하여 트라이의 모든 비어있는 노드를 제거하는 방식이다. 우선순위 트라이 구조는 이진 트라이 구조의 단점인 비어있는 노드를 제거하여 트라이의 깊이를 줄이고, 검색 성능을 향상시켰으나 프리픽스의 분포에 따라 불균형적인 구조가 되는 것을 피할 수는 없다.

이진 검색 트리 구조는 프리픽스들을 크기별로 정렬하여 이진 검색을 수행하는 구조이다. 그러나 앞서 설명한 바와 같이 프리픽스들 간의 네스팅 관계 때문에 이

진 검색 트리가 불균형 구조를 갖게 된다. 본 논문에서 제안하는 구조는 이진 검색 트리에 기초한 구조이므로 이진 검색 트리에 대하여 자세히 알아보고자 한다.

### 2.1 프리픽스의 크기에 따른 정렬

이진 검색 트리 알고리즘[2]에서는 길이가 다른 프리픽스들의 크기 비교를 위하여 다음과 같은 정의를 사용하였다.

*두 프리픽스 A와 B의 길이를 각각 n, m이라 할 때,*

***정의 1) 비교(Compare)***

*두 프리픽스의 길이가 같은 경우 (n=m), A와 B 의 수학적 (numerical) 값이 비교된다. 두 프리픽스의 길이가 다른 경우 ($n>m$), 짧은 프리픽스의 길이까지만 비교한다. 만약 두 프리픽스의 서브 스트링(sub-string)이 같은 경우, 긴 프리픽스의 m+1 번째 비트가 1이면 $A>B$, 아니면 $A<B$이다.*

*예) A=100, B=101: $B>A$*

*예) A=11011, B=110: $A>B$*

*예) A=11011, B=101: $A>B$*

***정의2) 매치(Match)***

*두 프리픽스가 같거나 짧은 프리픽스 길이까지가 같은 경우, 매치한다. 그렇지 않으면 두 프리픽스는 매치하지 않는다.*

*예) A=11011, B=110: A와 B는 매치한다.*

*예) A=11011, B=101: A와 B는 매치하지 않는다.*

***정의3) 디스조인트(Disjoint)***

*두 프리픽스가 매치하지 않으면 A와 B는 디스조인트이다.*

*예) A=11011, B=101: A와 B는 디스조인트이다.*

***정의4) 인클로저(Enclosure)***

*A를 서브 스트링으로 갖는 다른 프리픽스가 하나라도 존재하면 A는 인클로저이다.*

*예) A=110, B=11011: A는 B의 인클로저이다.*

### 2.2 네스팅 관계를 무시한 균형 트리 구조

표 1의 프리픽스 셋을 예로 하여 네스팅 관계를 무시하고 균형 트리를 구성하면 그림 2와 같다. 이 트리의 구성 방법은 다음과 같다. 앞서 설명한 정의 1을 사용하여 프리픽스들을 크기 순으로 정렬하면 그 중 가운데 프리픽스가 루트 노드가 된다. 이 노드의 왼쪽 서브 트리는 이 프리픽스보다 크기가 작은 프리픽스들을 갖게 되며, 오른쪽 서브 트리는 루트 노드의 프리픽스보다 크기가 큰 프리픽스들을 갖게 된다. 서브 트리에 속한 프리픽스들 중 가운데 프리픽스가 이 서브 트리의 루트 노드가 된다. 이 서브 트리의 루트 노드에 대하여 다음 서브 트리를 같은 방식으로 구성한다. 이와 같은 방식으로 모든 프리픽스를 트리의 노드에 저장한다.

그림 2에서 보듯이 네스팅 관계를 무시하고 크기별 정렬에 의한 균형 트리를 구성한 경우, 트리의 깊이를 최적화할 수 있다. 그러나 이와 같이 네스팅 관계를 무시하고 단지 검색 속도 향상을 위해 균형 트리를 구성한 경우, LPM을 수행하지 못해 잘못된 검색 결과가 나올 수 있다. 한 예로, 목적지 주소가 1101111인 패킷에 대한 주소 검색 결과, 이 트리 구조의 검색 과정에서는 매치하는 프리픽스가 없어 출력 포트를 결정할 수 없게 된다. 그러나 이 예시 셋에서 입력 패킷의 목적지 주소와 매치하는 가장 긴 프리픽스는 1101*로 출력 포트는 P5가 되어야 한다. 이는 110110*에 해당하는 노드와 입력 패킷의 목적지 주소의 크기를 비교하면서 1101*가 있음에도 불구하고 110110* 노드를 거치면서 1101*가 포함된 왼쪽 서브 트리는 검색 범위에서 제외되기 때문이다. 이와 같은 구조는 네스팅 관계에 있는 프리픽스들을 처리하지 못해, 검색 과정에서 입력 패킷의 목적지 주소와 매치할 가능성이 있는 프리픽스들을 제외시켜 잘못된 검색 결과를 가져온다. 따라서 트리의 깊이를 줄이기 위해 단순히 네스팅 관계를 무시하고 균형 트리를 구성하는 방법은 IP 주소 검색을 위한 LPM을 수행하지 못함을 알 수 있다.

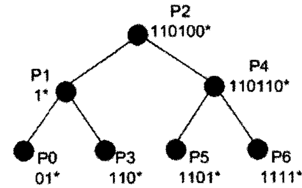



그림 2 네스팅 관계를 무시한 균형 트리 예시

### 2.3 이진 검색 트리(Binary Search Tree)

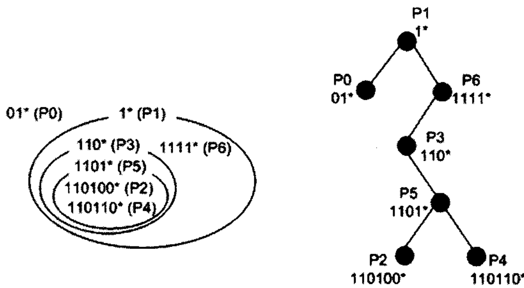
정확한 LPM을 수행하기 위해선 네스팅 관계를 고려한 트리를 구성하여 입력 패킷의 목적지 주소와 매치할 가능성이 없는 프리픽스들만을 제외시켜야 한다. 이를 해결하는 기존의 방식은 어떤 프리픽스들이 네스팅 관계에 있을 때, 인클로저 (정의 4)를 부-스트링(sub-string)으로 갖는 인클로즈드 프리픽스들을 인클로저의 서브 트리에 위치시키는 것이다. 따라서 검색 과정에서 검색 범위를 줄여가면서도 패킷의 목적지 프리픽스와 매치하는 모든 프리픽스들이 검색 범위에 남아 있게 된다.

그림 3(a)는 표 1의 프리픽스들의 네스팅 관계를 정리한 결과이다. 그림에서 네스팅 관계가 있는 프리픽스들은 원으로 둘러 싸여 있다. 즉, 인클로저는 인클로즈드 프리픽스들을 원으로 둘러싼다. 여기서 같은 원에 속하지 않는 프리픽스들은 서로 디스조인트 (정의 3)함을 의미한다.

이러한 네스팅 관계를 고려하여 이진 검색 트리를 구성하는 방법은 다음과 같다.먼저 디스조인트 프리픽스와 첫번째 레벨의 인클로저 프리픽스만을 크기별로 정렬하여 가운데 속한 프리픽스를 루트 노드로 선정한다. 이 예에서는 01*와 1*의 두개의 프리픽스 만이 이에 해당

되므로, 임의로 두번째 프리픽스를 루트 노드로 선정하였다. 인클로져에 속한 인클로즈드 프리픽스들은 인클로저가 노드로 선정되었을 때 정렬리스트에 포함되게 된다. 1*가 선정이 되었으므로, 다음 레벨에 속한 디스조인트 프리픽스인 1111*와 인클로져 프리픽스인 110*가 크기에 따라 정렬되어 리스트에 추가 되는데, 루트 노드인 1* 보다 모두 큰 값을 가지므로, 1*의 오른쪽 리스트에 추가된다. 1*의 왼쪽 리스트에는 01*만이 존재하므로, 01*는 왼쪽 서브 트리의 루트 노드로 선정이 되고, 오른쪽 리스트에는 110*와 1111*가 존재하는데, 임의로 1111*를 오른쪽 서브트리의 루트노드로 선정하였다. 이러한 과정을 모든 프리픽스가 선정이 될 때까지 반복하면 그림 3(b)와 같은 이진 검색 트리가 완성된다.

그림 3(b)에서 보듯이 예시 프리픽스 셋은 프리픽스 간에 네스팅 관계가 많아 트리의 불균형도가 매우 심함을 알 수 있다. 이와 같은 방식으로 트리의 네스팅 관계를 처리하면, 프리픽스들 간에 네스팅 관계가 많을수록 불균형도가 심해져 검색 속도를 더욱 저하시키게 된다.




(a) 프리픽스들의 네스팅 관계 예시 (b) BST 예시

그림 3 이진 검색 트리

가중 이진 검색 트리(Weighted Binary Search Tree) [3]는 같은 방식으로 트리의 네스팅 관계를 처리하되, 각 레벨의 루트를 선정함에 있어 가중치가 높은 인클로져를 먼저 선택하는 방식으로 이진 검색 트리의 불균형도를 완화하였다. 여기서 가중치는 자신을 부-스트링(sub-string)으로 하는 프리픽스의 개수에 따라 정하여진다. 디스조인트 프리픽스의 경우는 자신을 부-스트링으로 하는 다른 프리픽스가 존재하지 않으므로 가중치가 0이고, 인클로져 프리픽스는 자신의 백에 들어있는 프리픽스의 개수가 된다. 예를 들어 프리픽스 01*의 경우 가중치가 0이고, 1*의 경우 가중치가 5이다. 마찬가지로 프리픽스 110*의 경우 가중치가 3이고, 1111*의 경우 가중치가 0이다. 가중치가 높은 프리픽스를 먼저 선정하는 방식으로 트리를 구성하면 두번째 레벨의 루트 노드는 프리픽스 1111*가 아닌 프리픽스 110*가 선정이 되어, 전체적인 트리의 깊이가 감소한다. 그러나 가중 이진 검색 트리 역시 완전 균형 트리를 구성하지는 못한다.

## 3. 제안하는 검색 알고리즘

IP 주소 검색에선 프리픽스들의 길이가 다양하여, 입력 패킷의 목적지 주소와 가장 길게 매치하는 프리픽스를 찾아야 한다. 따라서 이 LPM을 수행하기 위해서 프리픽스들 간의 네스팅 관계를 고려해야 한다. 전 절에서 보았듯이 네스팅 관계를 고려하지 않은 균형 트리 구조는 잘못된 검색 결과를 가져온다. 또한 이를 해결하고자 인클로저의 서브 트리에 인클로즈드 프리픽스들이 위치하도록 하는 트리 구성 방식은 네스팅 관계가 많을수록 트리의 불균형도가 심해져 검색 속도를 저하시킨다.

본 논문에서 제안하는 알고리즘은 완전 균형 이진 검색 트리를 구성하되, 스위치 포인터를 사용하여 프리픽스들 간의 네스팅 관계를 처리하는 새로운 방식의 알고리즘이다.

### 3.1 라우팅 테이블 구성

프리픽스들 간의 네스팅 관계를 표현하기 위하여 제안하는 구조에서의 트리 노드는 노드에 저장된 프리픽스의 값 및 프리픽스의 길이, 출력 포트 정보에 더하여 추가적으로 두개의 정보를 저장하게 되는데, 스위치 포인터와 인클로져의 길이가 그것이다. 여기서 스위치 포인터란 각 노드에 저장된 프리픽스의 다이렉트 인클로저가 저장된 메모리의 주소를 말한다. 이 과정을 자세히 보면 다음과 같다.

앞의 그림 2의 예시에서 보았듯이 단순히 균형 트리를 구성한 결과, 입력 패킷의 목적지 주소와 매치할 가능성이 있는 프리픽스까지 검색 범위에서 제외시켜 잘못된 검색 결과를 가져옴을 알 수 있었다. 제안하는 구조에서는 완전 균형 트리를 구성하되, 입력 패킷과 일치할 가능성이 있는 프리픽스들의 정보를 갖게 하기 위하여 각 프리픽스는 자신의 다이렉트 인클로저를 가르키는 포인터(EncPtr)와 인클로져의 길이(EncLen) 값을 갖게 하였다. 여기서 다이렉트 인클로저란 자신의 인클로저들 중 가장 긴 프리픽스를 말한다. 예를 들어 표 1의 프리픽스 셋에 대하여 각 프리픽스의 스위치 포인터를 나타내면 다음과 같다. 01*와 1*는 인클로저가 없으므로 스위치 포인터는 NULL 값을, 인클로져 길이는 0의 값을 가지며, 110*는 1*를 인클로저로 가지므로 스위치 포인터는 1* 가 저장된 엔트리의 메모리 주소이고, 인클로져 길이는 1이다. 또한 110100*는 인클로저로 1*, 110*를 가지나 다이렉트 인클로저는 110*이므로 스위치 포인터는 110*이 저장된 엔트리의 주소이고 인클로져 길이는 3이 된다. 이와 같은 방식으로 표 1의 예시 프리

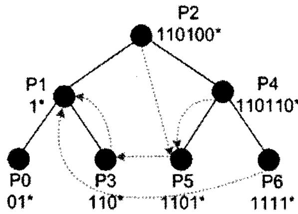


그림 4 제안된 균형 트리 구조

표 2 예시 프리픽스 셋에 대한 라우팅 테이블

| | Prefix | PreLen | EncPtr | EncLen |
|---|---|---|---|---|
| 0 | 01* | 2 | - | 0 |
| 1 | 1* | 1 | - | 0 |
| 2 | 110* | 3 | 1 | 1 |
| 3 | 110100* | 6 | 4 | 4 |
| 4 | 1101* | 4 | 2 | 3 |
| 5 | 110110* | 6 | 4 | 4 |
| 6 | 1111* | 4 | 1 | 1 |

| Prefix | Length | Out Port | EncPtr | EncLen |
|---|---|---|---|---|

그림 5 제안된 알고리즘의 엔트리 구조

픽스 셋에 대해 각 프리픽스의 다이렉트 인클로저로의 포인터를 나타내어 균형 트리 구조를 그리면 그림 4와 같다. 표 2에 그림 4를 토대로 구성한 제안하는 구조의 라우팅 테이블을 보였다. 제안하는 알고리즘의 엔트리 구조는 그림 5와 같은데, 각 엔트리는 프리픽스 값, 프리픽스의 길이, 출력포트 정보, 스위치 포인터, 인클로저 길이 값을 갖는다.

### 3.2 IP 주소 검색

제안하는 구조의 검색 방법은 기존의 이진 검색 방법 즉, 엔트리에 저장된 프리픽스 정보만을 사용하여 검색 범위를 좁혀 나가는 방법과는 다르다. 제안된 구조는 인클로저의 서브 트리에 그것의 인클로즈드 프리픽스들이 있다고 보장하지 않으므로, 기존의 이진 검색만으로는 입력 패킷과 매치할 가능성이 있는 프리픽스들이 검색 범위에서 제외될 수 있어 잘못된 검색 결과를 가져온다. 제안하는 알고리즘에서의 IP 주소 검색은 다음과 같은 두 단계로 이루어져 있다.

첫 번째 단계는 이진 검색의 단계로서, 이진 검색 순서에 따라 트리의 리프에 도달할 때까지 가장 길게 매치하는 프리픽스의 출력 포트를 찾아냄과 동시에 프리픽스와 일치하지 않더라도 이것의 다이렉트 인클로저의 정보를 이용하여 이것의 인클로저들과 매치 가능성이 있다면 이 정보를 기억하게 한다.

두 번째 단계는 스위치 포인터 검색의 단계로서, 이진 검색의 결과 현재까지 가장 길게 일치한 프리픽스보다 더 길게 일치할 가능성이 검색한 범위 밖에 있는지 판단하여 기억된 스위치 포인터를 따라가면서 검색을 진행하여 가장 길게 매치하는 프리픽스의 출력 포트를 찾는다.

3.2.1 이진 검색

이 검색 과정의 비교 엔트리 순서는 이진 검색 순서를 따른다. 즉, 현재 검색 범위에서 가운데 엔트리의 프리픽스와 크기를 비교하여 작으면 현재 검색 범위 중 이 엔트리의 윗쪽, 크면 아랫쪽이 다음 검색 범위로 좁혀 진다. 그러나 제안하는 구조는 기존의 현재 엔트리의 프리픽스 정보만을 이용해 이진 검색을 하던 방식과는 차이가 있다. 즉, 제외된 검색 범위에서도 입력 패킷의 목적지 주소와 더 길게 매치하는 프리픽스가 존재 할 가능성이 있으므로 이 정보까지도 찾아내야 한다.

제안하는 구조에서는 검색과정에서 하나의 엔트리에 접근할 때마다 네 종류의 정보가 갱신이 되는데, 현재까지 가장 길게 일치한 프리픽스(best matching prefix, BMP), 현재까지 가장 길게 일치한 프리픽스의 길이(best matching length, BML), 현재까지 거쳐온 각 프리픽스의 인클로저와 매치하는 가장 긴 길이(matching enclosure length, MEL) 및 인클로저로 가는 스위치 포인터(Switch Pointer, SP)가 그것이다.

이진 검색 순서에 따라 엔트리를 찾아가되, 각 엔트리에서의 수행되는 동작은 다음과 같다. 첫째로, 기존의 이진 검색 과정과 같이 입력 패킷과 현재 엔트리의 프리픽스가 일치하는 경우, BML보다 더 길게 매치하면 이 프리픽스의 출력 포트와 길이로 각각 BMP와 BML를 업데이트한다. 둘째로, 현재 엔트리의 프리픽스와 매치하지 않더라도, 입력 패킷의 목적지 주소가 이 프리픽스의 인클로저의 길이 내에서 매치하는 길이(local matching enclosure length, LMEL)를 계산하여, 이 길이가 MEL보다 길면, 스위치 포인터와 MEL을 현재 엔트리의 EncPtr 및 LMEL로 업데이트한다.

제안하는 구조의 첫번째 검색 단계인 이진 검색이 마무리 되면, BMP, BML는 검색 범위에서 현재까지 가장 길게 매치한 프리픽스의 출력 포트와 이 프리픽스의 길이 정보를 갖게 된다. 또한 스위치 포인터는 검색 범위에 포함되지 않았던 곳에 있으나 입력 패킷의 목적지 주소와 가장 길게 매치할 가능성이 있는 프리픽스의 정보를 가지게 된다. MEL는 검색 범위에 포함되지 않았던 곳에서 매치할 가능성이 있는 가장 긴 길이 정보이다.

최종적으로 기억하는 BML와 MEL의 정보를 이용하여 검색의 두번째 단계인 스위치 포인터 검색으로 진행할 것인지를 판단하는데, MEL가 BML보다 길면 제외된 검색 범위에서 현재까지 검색된 프리픽스보다 더 길게 매치할 프리픽스가 있을 가능성이 있으므로 스위치

포인터 검색을 진행한다. 그렇지 않으면 BMP를 최종 검색 결과로 결정한다. 이진 검색(BS_Search)의 슈도 코드는 다음과 같다.

```
BML=0; BMP=default; MEL=0;SP=default;
Current_Range= from the 1st entry to the last entry in
the table
BS_Search (input, BML, BMP, MEL, SP,
Current_Range)
  m = the index of medium entry of the Current range;
  if (table[m].Prefix match with input) & (table[m].
    PreLen > BML)
         Update BMP, BML;
  Find LMEL;
  if LMEL > MEL
    Update MEL, SP;

  if input < table[m].Prefix
    Current_Range = from the 1st entry to the (m-1)st
    entry in the Current_Range;
  else
    Current_Range = from the (m+1)st entry to the
    last entry in the Current_Range;

  if the Current_Range is empty
    if MEL> BML
      SP_Search (input, BML, BMP, MEL, SP);
    else
      return BMP;
  else BS_Search (input, BML, BMP, MEL, SP,
  Current_Range) ;
```

3.2.2 스위치 포인터 검색

앞에서 설명하였듯이 제안된 구조는 인클로저의 서브 트리에 그것의 인클로즈드 프리픽스들을 가지도록 하지 않는다. 따라서 기존의 이진 검색만으로는 입력 패킷과 매치할 가능성이 있는 프리픽스들이 검색 범위에서 제외되어 잘못된 검색 결과를 가져온다. 따라서 제외된 검색 범위에서도 이진 검색 결과, 현재까지 가장 길게 매치한 프리픽스의 길이(BML)보다 길게 매치할 프리픽스가 있을 가능성이 있다면 이 범위에 대한 검색을 진행해야 한다. 따라서 스위치 포인터 검색은 이 제외된 검색 범위 중에서도 매치 가능성이 높은 소수의 프리픽스만을 가지고 검색을 진행한다. 즉, 위의 이진 검색에서 지속적으로 갱신된 스위치 포인터(SP)와 현재까지 거쳐온 각 프리픽스의 인클로저와 매치하는 가장 긴 길이(MEL)의 정보를 사용하여 검색을 하게 된다.

각 엔트리에서의 검색 과정은 다음과 같다. 스위치 포인터를 인덱스로 하는 엔트리로 가서 이 엔트리의 프리픽스와 입력 패킷의 목적지 주소가 매치하는지 판단한다. 첫번째 경우로, 매치하지 않는 경우, 이 프리픽스의 인클로저 길이 내에서 매치하는 길이 (LMEL)가 BML보다 크다면 이 프리픽스의 인클로저가 BML 보다 더 길게 매치할 가능성이 있다. 따라서 이 프리픽스의 EncPtr를 인덱스로 하는 엔트리로 가서 검색을 반복한다. 두번째 경우로, 매치한다면 이 프리픽스가 가장 길게 매치하는 프리픽스 이므로 더 이상 포인터를 따라가지 않고, 현재까지의 BML과 이 프리픽스의 길이를 비교하여, 이 프리픽스의 길이가 BML보다 긴 경우, BML 값을 갱신하고, BMP도 현재 프리픽스의 출력 포트로 갱신한 후 검색을 종료한다. 스위치 포인터 검색의 슈도 코드는 다음과 같다.

```
SP_Search(input, BML, BMP, SP)
if (table[SP].Prefix match with input) &
(table[SP].PreLen > BML)
   Update BMP, BML;
else begin
   Find LMEL;
   if (table[SP].EncLen!=0) & (LMEL > BML) begin
      SP = table[SP].EncPtr;
      SP_Search(input, BML, BMP, SP);
   end
end
```

제안하는 구조에서의 검색의 예로 입력 패킷의 목적지 주소가 1101000인 경우, 110100* 프리픽스를 가지는 엔트리로부터 검색을 시작하여 입력 패킷의 목적지 주소가 프리픽스와 일치하므로, BMP=P2, BML=6, MEL=4, SP=4가 된다. 그 다음, 1*를 프리픽스로 하는 엔트리에서의 검색 결과, BML 보다 짧게 매치하므로 BMP, BML는 업데이트되지 않는다. 또한 이 프리픽스의 다이렉트 인클로저가 없으므로 MEL과 SP도 업데이트되지 않는다. 마지막으로 110*를 프리픽스로 하는 엔트리에서의 검색 결과, 아무것도 업데이트되지 않으므로 최종적으로 BMP=P2, BML=6, MEL=4, SP=4 가 된다. 더 이상 검색할 엔트리가 없으므로 BML와 MEL을 비교한다. 그 결과 BML가 더 크므로 제외된 검색 범위에서 더 길게 매치할 프리픽스가 없음을 알게 된다. 따라서 스위치 포인터 검색을 진행하지 않고 검색을 종료한다. 최종 검색 결과는 출력 포트 P2이다.

또 다른 예로 입력 패킷의 목적지 주소가 1101111인 경우, 루트 노드에서 110100*과 비교하여 매치하지 않으므로, BMP=NULL, BML=0이고, 4비트 인클로저와는 일치하므로 MEL=4, SP=4가 된다. 입력 패킷이 프리픽스 110100보다 크므로 오른쪽 서브 트리로 진행하여 110110*와 비교되고 매치하지 않으므로, BMP와 BML은 업데이트 되지 않는다. 또한 LMEL이 4로서 현재의 MEL과 값이 같으므로 MEL과 SP는 업데이트 되지 않

는다. 마지막으로 다음 노드인 1111*와 비교하여 MEL이 LMEL인 1보다 크므로 더 이상 업데이트 되지 않고, 최종 이진 검색 결과는 BMP=NULL, BML=0, MEL=4, SP=4이다. MEL가 BML보다 크므로 제외된 검색 범위에서 더 길게 매치할 프리픽스가 있을 가능성이 있다. 따라서 스위치 포인터 검색을 진행한다. 이진 검색 결과의 SP가 4이므로 인덱스가 4인 엔트리로 간다. 이 엔트리의 프리픽스는 길이가 4인 1101*로 1101111과 매치하므로 더 이상 다른 엔트리에서의 검색을 하지 않고 BMP=P5, BML=4가 된다.

### 3.3 업데이트

3.3.1 삽입

제안된 구조는 각 프리픽스의 다이렉트 인클로져 정보를 가지고 있어야 한다. 따라서 기존의 라우팅 테이블에 새로운 프리픽스를 삽입하려고 할 때, 이 프리픽스의 다이렉트 인클로져의 정보 뿐 아니라, 기존의 프리픽스들 중 이 프리픽스를 다이렉트 인클로져로 갖게 되는 것들을 찾아 새로운 정보로 업데이트시켜야 한다. 따라서 새로운 프리픽스의 삽입하는 경우, 다음과 같이 크게 2가지의 작업이 요구된다. 첫째로 삽입하려는 프리픽스가 현재 엔트리의 프리픽스의 인클로져인 경우, 삽입 프리픽스가 현재 엔트리의 프리픽스의 다이렉트 인클로져가 된다면 이 엔트리의 다이렉트 인클로져 정보를 삽입 프리픽스로 업데이트시켜야 한다. 둘째로 삽입하려는 프리픽스가 현재 엔트리의 프리픽스의 인클로즈드 프리픽스인 경우, 삽입 프리픽스의 다이렉트 인클로져 정보를 이 엔트리의 프리픽스로 업데이트시킨다.

위에서 보듯이 하나의 프리픽스를 삽입하기 위해선 이 프리픽스를 다이렉트 인클로져로 갖는 프리픽스들과 삽입 프리픽스의 다이렉트 인클로져가 되는 프리픽스를 찾아야 한다. 그러나 기존의 라우팅 테이블의 모든 엔트리를 검사하여 이 프리픽스들을 갖는 엔트리들을 찾아내 업데이트시키는 것은 현실적이지 않다. 따라서 가능한 적은 수의 엔트리를 검사하여 필요한 모든 정보를 업데이트시켜야 한다. 즉, 업데이트해야 할 엔트리를 포함하는 가능한 작은 범위를 정의해야 한다. 삽입하려는 프리픽스를 범위로 표현하여 이 범위 내에서 먼저 업데이트할 엔트리들을 찾는다. 즉, 삽입 프리픽스에 각각 '0'과 '1'을 붙여 프리픽스 최대 길이로 확장시킨 것을 이 삽입 프리픽스의 범위의 시작점, 끝점으로 한다. 예를 들어, 삽입 프리픽스가 110*이며 IP 주소 길이를 7비트라고 가정하면, 이 프리픽스의 범위는 (1100000, 1101111)가 된다. 이렇게 확장시킨 첫번째 스트링을 가지고 각각 라우팅 테이블에서 다음 검색 범위가 현재 엔트리의 아래가 되지 않을 때까지 이진 검색 순서를 따라간다. 이 과정에서 각 엔트리를 거칠 때마다 삽입 프리픽스와 현재 엔트리의 프리픽스가 매치하면 이 엔트리를 검사할 범위의 시작점으로 업데이트한다. 마찬가지로 확장시킨 두번째 스트링에 대해서는 다음 검색 범위가 현재 엔트리의 위가 되지 않을 때까지 이진 검색 순서를 따라간다. 마찬가지로 각 엔트리를 거칠 때마다 삽입 프리픽스와 현재 엔트리의 프리픽스가 매치하면 이 엔트리를 검사할 범위의 끝점으로 업데이트한다. 그 결과, 검사할 범위는 이렇게 찾은 시작점에서 끝점까지가 된다. 이 범위는 항상 삽입 프리픽스의 모든 인클로즈드 프리픽스들을 포함한다. 따라서 위에서 언급한 첫번째 업데이트의 경우 즉, 삽입 프리픽스를 다이렉트 인클로져로 갖는 프리픽스들은 이 범위 내에 반드시 포함된다. 그러나 이 범위 내에는 삽입 프리픽스의 인클로져들을 포함하지 못하므로 두번째 업데이트의 경우 즉, 삽입 프리픽스의 다이렉트 인클로져의 정보는 이 범위 내에 반드시 포함된다고 할 수 없다. 만약에 이 범위 내에서 삽입 프리픽스의 가장 긴 인클로져의 정보를 찾았다면 삽입 프리픽스의 가장 긴 인클로져이므로 그것이 전체 엔트리 중에서 삽입 프리픽스의 다이렉트 인클로져가 된다.

검사할 범위를 찾았다면 그 범위 내의 모든 엔트리들을 거치면서 그 엔트리의 다이렉트 인클로져 정보를 업데이트 시키거나 삽입 프리픽스의 다이렉트 인클로져 정보를 찾는다. 따라서 위에서 언급한 두가지 경우로 크게 나누어 볼 수 있다. 첫째로 삽입 프리픽스를 인클로져로 갖는 프리픽스인 경우, 현재 엔트리의 다이렉트 인클로져보다 삽입 프리픽스가 더 길면 이 엔트리의 다이렉트 인클로져를 삽입 프리픽스로 업데이트 시킨다. 둘째로 삽입 프리픽스를 인클로즈드 프리픽스로 하는 프리픽스인 경우, 이 엔트리의 프리픽스가 현재까지 업데이트시킨 삽입 프리픽스의 다이렉트 인클로져보다 더 길면 삽입 프리픽스의 다이렉트 인클로져를 이 엔트리의 프리픽스로 업데이트시킨다.

이 범위 내에서 삽입 프리픽스의 다이렉트 인클로져 정보를 업데이트시키지 못하였다면 이 범위 밖에 이 정보가 있을 가능성이 있다. 따라서 삽입 프리픽스의 길이를 하나씩 줄여가면서 매치하는 프리픽스를 찾는다. 찾았다면 이것을 삽입 프리픽스의 다이렉트 인클로져로 업데이트하고 인클로져 검색을 마친다.

이 모든 과정을 거친 후에는 삽입 프리픽스를 프리픽스로 하는 엔트리를 테이블의 해당 자리에 삽입시킨다. 이와 같은 프리픽스 삽입의 슈도 코드는 다음과 같다.

```
Insert(prefix)
input.Prefix=prefix; input.EncPtr = NULL; input.EncLen
= NULL;
```

```
prefix(S) = the prefix expanded to the maximum length
by the padding with '0'
prefix(E) = the prefix expanded to the maximum length
by the padding with '1'
Find out the Start and the End of the checking range
using prefix(S) & prefix(E)
for i=Start; i<=End; i=i+1 begin
  if input is the enclosure of table[i].Prefix begin
    if the Length of input.Prefix > table[i].EncLen
      Update table[i].EncPtr, table[i].EncLen
  end
  if input is the enclosed prefix of table[i].Prefix begin
    if the Length of input.Prefix > input.EncLen
      Update input.EncPtr, input.EncLen
  end
end
i=1;
while(input.EncPtr=NULL)&(i<n) begin
  Current_Range= from the 1st entry to the last entry
  in the table;
  m = the index of medium entry of the Current range;
  while(Current_Range is not empty) & (input.EncPtr =
  NULL) begin
    if table[m].Prefix match with input.Prefix[0: the
    Length of input.Prefix-1-i]
      Update input.EncPtr, input.EncLen

    if input.Prefix[0: the Length of input.Prefix-1-i] <
    table[m].Prefix
      Current_Range = from the1st entry to the (m-1)st
      entry in the Current_Range;
    else
      Current_Range = from the (m+1)st entry to the
      last entry in the Current_Range;
  end
end
Insert input in the routing table
```

예를 들어, 삽입하고자 하는 프리픽스가 11*일 때, 1100000, 1111111로 확장시킨 스트링들로 테이블에서 검사할 범위를 찾는다. 1100000에 대하여 예를 들면, 먼저 표 2의 테이블의 가운데인 3번 엔트리로 간다. 11*와 매치하므로 시작점(Start)을 3으로 업데이트한다. 다음, 1100000이 현재 엔트리의 프리픽스보다 작으므로 위의 엔트리들 중 가운데인 1번 엔트리로 간다. 1*와 매치하므로 시작점(Start)을 1로 업데이트한다. 다음, 1100000이 현재 엔트리의 프리픽스 1*보다 크므로 시작점검색을 마친다. 그 결과, 검사할 범위의 시작점은 1번 엔트리가 된다. 마찬가지로 1111111 스트링을 가지고 끝점을 검색하면 6번 엔트리가 된다. 따라서 검사할 범위는 1번 엔트리에서 6번 엔트리까지이다. 먼저 1번 엔트리에 대하여 업데이트를 시작한다. 삽입 프리픽스 11*는 1*의 인클로즈드 프리픽스이고 현재까지 삽입 프리픽스의 다이렉트 인클로저 정보가 없었으므로 이것의 다이렉트 인클로저의 포인터와 길이를 각각 1* 엔트리, 1로 업데이트시킨다. 다음 2번 엔트리에 대해선 삽입 프리픽스가 110*의 인클로저이며 삽입 프리픽스가 현재 이 엔트리의 다이렉트 인클로저보다 길기 때문에 이 엔트리의 다이렉트 인클로저의 포인터와 길이를 각각 삽입 프리픽스의 엔트리, 2로 업데이트시킨다. 3,4,5번 엔트리에 대해선 삽입 프리픽스가 각 엔트리의 프리픽스의 인클로저이나 이 엔트리의 다이렉트 인클로저 길이가 삽입 프리픽스보다 기므로 아무것도 업데이트하지 않는다. 6번 엔트리에 대해선 2번 엔트리와 같이 엔트리의 다이렉트 인클로저의 포인터와 길이를 각각 삽입 프리픽스의 엔트리, 2로 업데이트시킨다.

3.3.2 삭제

기존의 라우팅 테이블에서 하나의 프리픽스를 삭제하고자 할 때, 이 프리픽스를 가지는 엔트리를 삭제해야 할 뿐 아니라, 이 프리픽스가 다이렉트 인클로저인 프리픽스들에 대하여 다이렉트 인클로저 정보를 업데이트시켜야 한다. 따라서 이 경우에 해당하는 엔트리들을 찾아 그들의 다이렉트 인클로저를 삭제하고자 하는 프리픽스의 다이렉트 인클로저로 바꿔주면 된다. 그러나 라우팅 테이블의 모든 엔트리를 검사하여 이 엔트리들을 찾아내 업데이트시키는 것은 현실적이지 않다. 따라서 가능한 적은 수의 엔트리를 검사하여 필요한 모든 정보를 업데이트시키기 위하여 프리픽스 삽입에서와 같은 방식을 적용한다. 즉, 삭제하고자 하는 프리픽스에 각각 '0'과 '1'을 붙여 프리픽스 최대 길이로 확장시킨 것으로 테이블에서 검사할 범위의 시작점과 끝점을 찾는다. 이 범위 내에 삭제하고자 하는 프리픽스의 모든 인클로즈드 프리픽스들이 포함되므로 업데이트할 엔트리들 역시 이 범위 내에 모두 포함된다. 따라서 삽입의 경우와 같이 이 범위 밖을 검사해야 하는 경우는 존재하지 않는다.

검사할 범위를 찾았으면 각 엔트리를 거치면서 현재 엔트리의 프리픽스가 삭제하려는 프리픽스를 다이렉트 인클로저로 가진다면 이 엔트리의 다이렉트 인클로저 정보를 삭제하려는 프리픽스의 다이렉트 인클로저 정보로 바꿔준다.

이와 같은 프리픽스 삭제의 슈도 코드는 다음과 같다.

```
Delete(input)
Find out the entry whose prefix is input
prefix(S) = the prefix expanded to the maximum length
by the padding with '0'
prefix(E) = the prefix expanded to the maximum length
by the padding with '1'
Find out the Start and the End of the checking range
using prefix(S) & prefix(E)

for i=Start; i<=End; i=i+1 begin
```

```
  if input is the encloser of table[i].Prefix begin
    if the Length of input = table[i].EncLen
      Update table[i].EncPtr, table[i].EncLen
  end
end
Delete the entry whose prefix is input
```

예를 들어 삭제하고자 하는 프리픽스가 110*인 경우, 먼저 110*를 프리픽스로 갖는 2번 엔트리의 다이렉트 인클로저 정보를 기억한다. 다음 110*를 다이렉트 인클로저로 갖는 엔트리들의 정보를 업데이트 시켜야 한다. 따라서 프리픽스의 삽입 경우와 같은 방식으로 검사할 범위를 찾는다. 그 결과, 2번에서 5번 엔트리가 되며 2번 엔트리는 삭제할 엔트리이므로 검사를 제외시키고 3번부터 검사를 시작한다. 3번 엔트리의 프리픽스 110100*는 110*를 다이렉트 인클로저로 가지고 있지 않으므로 업데이트하지 않는다. 4번 엔트리의 프리픽스 1101*는 110*를 다이렉트 인클로저로 가지므로 이 엔트리의 다이렉트 인클로저 포인터와 길이를 각각 110*의 다이렉트 인클로저 포인터와 길이인 1* 엔트리, 1로 업데이트 한다. 5번 엔트리의 경우는 3번 엔트리의 경우와 마찬가지로 아무것도 업데이트하지 않는다. 검색 범위의 모든 엔트리에 대하여 업데이트를 마쳤으므로 마지막으로, 110* 엔트리를 삭제한다.

## 4. 시뮬레이션 결과 및 성능 평가

본 논문에서는 제안된 알고리즘의 성능을 분석하기 위해, 2006년 5월에 실제 백 본 라우터로부터 내려 받은 몇 개의 데이타베이스 W2, AADS, W1, E1, PORT80, GROUPTLCOM, TELSTRA에 대해 알고리즘을 수행하였다[7]. 그 결과를 비교, 분석한 결과는 표 3과 같다. 표 3에서 $T_{avg}$는 평균 메모리 접근 횟수, $T_{max}$는 최대 메모리 접근 횟수, $T_{min}$은 최소 메모리 접근 횟수를 의미한다. 또한 $x$는 라우팅 테이블의 인덱스와 EncPtr에 할당하는 bit 수로서, 제안하는 구조는 프리픽스 듀플리케이션(duplication)이 일어나지 않기 때문에 엔트리 수가 프리픽스 수와 같으므로, N을 프리픽스의 개수라고 할 때 $x = \lceil \log_2 N \rceil$ 이다.

주소 검색 알고리즘의 성능 평가에 있어서 중요한 기준은 검색 속도(search speed)와 메모리 요구량(storage requirement)이다. 그러나 인터넷 사용자가 급증함에 따라 라우팅 테이블의 사이즈가 커져 링크 속도로 패킷을 처리하지 못하고 있다. 따라서 효율적인 주소 검색 알고리즘의 개발을 위해 중점을 두어야 할 기준은 검색 속도라 할 수 있다. 메모리 접근 횟수는 이 패킷을 처리하는 속도를 평가하는 데 있어서 매우 중요한 수치이다. 따라서 제안하는 구조의 성능을 평가하기 위해 표 4와 그림 6에서 제안하는 구조와 여러 알고리즘들의 평균 메모리 접근 횟수를 비교한다. 표에서 보듯이 제안하는 구조의 메모리 접근 횟수는 LC-trie와 PV-trie보다는 크고 BSR과는 비슷하다. 그러나 다른 알고리즘들보다 메모리 접근 횟수가 상대적으로 작다. 또한 포워딩 테이블의 사이즈가 커짐에 따라 검색속도가 크게 저하되지 않음을 볼 수 있어 확장성이 매우 우수한 구조라 할 수 있다.

앞에서 말했듯이 주소 검색 알고리즘의 성능 평가를 위해 두번째로 중요한 기준은 메모리 요구량이다. 표 5와 그림 8에서 프리픽스 당 메모리 요구량(Byte)을 비교하여 나타낸다. 이 수치는 아래 그림 7에서 보이는 엔

표 3 제안된 구조의 시뮬레이션 결과

| Input | *N* | #Input | #Entry | *x* | $T_{avg}$ | $T_{max}$ | $T_{min}$ | *Memory(Kbyte)* |
|---|---|---|---|---|---|---|---|---|
| W1 | 14553 | 43659 | 14553 | 14 | **13.88** | 16 | 13 | **127.91** |
| AADS | 20204 | 60612 | 20204 | 15 | **14.38** | 17 | 14 | **182.51** |
| W2 | 29584 | 88752 | 29584 | 15 | **14.91** | 17 | 14 | **267.24** |
| E1 | 39465 | 511802 | 39465 | 16 | **16.09** | 18 | 15 | **366.13** |
| PORT80 | 112310 | 336930 | 112310 | 17 | **16.87** | 22 | 16 | **1069.36** |
| Grouptlcom | 170601 | 511802 | 170601 | 18 | **17.49** | 22 | 17 | **1666.03** |
| Telstra | 227223 | 681669 | 227223 | 18 | **17.87** | 22 | 17 | **2218.97** |

표 4 여러 알고리즘들과 평균 메모리 접근 횟수 비교

| Input | **Prop** | B-trie [1] | BST [2] | WBST [3] | P-trie [4] | LC-trie [5] | BSR [6] | PV-trie [7] |
|---|---|---|---|---|---|---|---|---|
| W1 | **13.9** | 22.9 | 14.1 | 13.9 | 16.7 | 2.9 | 14.2 | 12.9 |
| W2 | **14.9** | 23.1 | 15.6 | 15.0 | 18.2 | 4.3 | 15.2 | 13.9 |
| E1 | **16.1** | 23.2 | 15.8 | 15.4 | 18.2 | 3.2 | 15.7 | 14.3 |
| PORT80 | **16.9** | 22.2 | 26.0 | 20.6 | 20.4 | 10.6 | 16.8 | 15.3 |
| Grouptlcom | **17.5** | 22.3 | 27.0 | 21.7 | 20.8 | 11.0 | 17.3 | 16.0 |
| Telstra | **17.9** | 24.6 | 30.8 | 23.9 | 22.9 | 11.2 | 17.6 | 16.5 |

표 5 여러 알고리즘들과 프리픽스 당 메모리 요구량(byte) 비교

| Input | Prop | B-trie | BST | WBST | P-trie | LC-trie | BSR | PV-trie |
|---|---|---|---|---|---|---|---|---|
| W1 | **9** | 31 | 10 | 10 | 10 | 440 | 12 | 24 |
| W2 | **10** | 22 | 10 | 10 | 10 | 159 | 12 | 23 |
| E1 | **10** | 26 | 10 | 10 | 10 | 213 | 12 | 24 |
| PORT80 | **10** | 12 | 10 | 10 | 10 | 65 | 9 | 15 |
| Grouptlcom | **10** | 11 | 10 | 10 | 10 | 48 | 9 | 15 |
| Telstra | **10** | 12 | 10 | 10 | 10 | 464 | 9 | 17 |

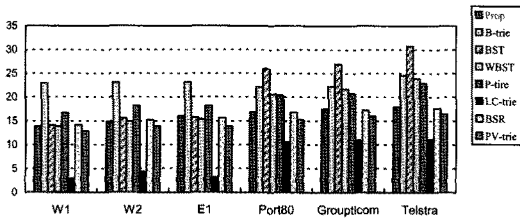



그림 6 여러 알고리즘들과 평균 메모리 접근 횟수 비교 그래프

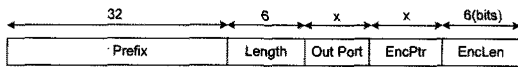



그림 7 엔트리에 할당한 bit 수

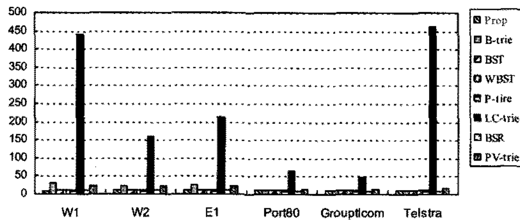



그림 8 여러 알고리즘들과 프리픽스 당 메모리 요구량 (Byte) 비교 그래프

트리 구조에서 각각에 할당한 bit 수를 토대로 계산한 결과이다. 즉, 프리픽스 당 메모리 요구량(byte)를 계산하는 식을 표현하면 $\{(44+2x)\times(\#Entry)\}/(\#Prefix\times8)=(44+2x)/8$가 된다. 표에서 보듯이 제안하는 구조는 다른 알고리즘들의 메모리 요구량과 비교하여 그 수치가 월등히 작거나 비슷한 수준이다. 또한 우수한 검색 속도를 보이는 LC-trie와 PV-trie보다 훨씬 좋은 메모리 요구량 수치를 보이고 있다.

## 5. 결 론

LPM에 의한 IP주소 검색 알고리즘의 성능 평가에 있어서 중요한 기준으론 첫째로, 검색 속도(search speed)이며 둘째로, 메모리 요구량(storage requirement)이다. 이를 위해 이진 트라이(binrary trie), P-트라이(priority trie), LC 트라이(LC trie), BST(binary search tree), WBST(weighted binary search tree)등의 여러 구조들이 제안되었으나 이들의 구조는 불균형(unbalanced) 구조이다. 이들 불균형 구조의 주요 원인은 네스팅 관계에 있는 프리픽스들 중 짧은 프리픽스인 인클로저가 긴 프리픽스인 인클로즈드 프리픽스들을 서브 트리에 갖게 하기 때문이다. 따라서 트라이(trie) 혹은 트리(tree)의 깊이(depth)가 비효율적으로 커져 최악의 경우(worst-case) 경우, 검색 속도가 매우 느려지는 단점이 있다.

제안하는 구조는 이진 검색 트리 구조의 불균형 문제를 해결하기 위해 완전 균형 이진 검색 트리를 구성하는 동시에 프리픽스들 간에 네스팅 관계가 있을 때 인클로저로 가는 스위치 포인터를 사용하여 네스팅 관계를 처리하는 새로운 접근 방식을 시도하였다. 그 결과, 제안하는 구조는 메모리 요구량을 늘리지 않으면서도 검색 속도에서의 우수한 향상을 보였다.

## 참 고 문 헌

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

[2] N. Yazdani and P. S. Min, "Fast and scalable schemes for the IP address lookup problem," *Proc. IEEE HPSR2000*, pp.83-92, 2000.

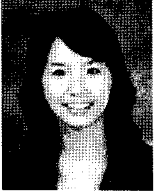
[3] C. Yim, B. Lee, and H. Lim, "Efficient binary search for IP address lookup," *IEEE Communications Letters*, vol.9, no.7, pp.652-654, Jul. 2005.

[4] H. Lim and J. Mun, "An efficient IP address lookup algorithm using a priority-trie," *Proc. IEEE Globecom2006*, pp.1-5, Nov. 2006.

[5] V. C. Ravikumar, R. Mahapatra, and J. Liu, "Modified LC-trie based efficient routing lookup," *Proc. MASCOTS2002*, pp.177-182, Oct. 2002.

[6] 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.

[7] http://www.potaroo.net



**김 형 기**

2007년 이화여자대학교 정보통신학과 졸업(학사). 2007년~이화여자대학교 전자공학과 석사과정. 관심 분야는 라우터나 스위치 등의 네트워크 관련 SoC 설계

**임 혜 숙**

1986년 서울대학교 제어계측공학과 졸업(학사). 1986년 8월~1986년 2월 삼성 휴렛 팩커드 연구원. 1991년 서울대학교 제어계측공학과 졸업(석사). 1996년 The University of Texas at Austin, Electrical and Computer Engineering 졸업(박사). 1996년 11월~2000년 7월 Lucent Technologies Member of Technical Staff. 2000년 7월~2002년 2월 Cisco Systems, Hardware Engineer. 2002년 3월~이화여자대학교 공과대학 전자공학과 부교수. 관심분야는 라우터나 스위치 등의 네트워크 관련 SoC 설계, TCP/IP 관련 하드웨어 설계