IP 어드레스 검색을 위한 새로운 pipelined binary 검색 구조

정회원 임 혜 숙*, 준회원 이 보 미, 정 여 진

A New Pipelined Binary Search Architecture for IP Address Lookup

Hye-Sook Lim* *Regular Member*,
Bo-Mi Lee, Yeo-Jin Jung *Associate Members*

요 약

라우터에서의 어드레스 검색은 일초에 수천만개 이상으로 입력되는 패킷에 대하여 실시간으로 처리되어야하기 때문에 인터넷 라우터는 효율적인 IP 어드레스 검색 구조를 갖도록 설계되어야 한다. 본 논문에서는 [1]에서 제안된 IP prefix의 binary tree에 기초한 효율적이면서 실용적인 IP 어드레스 검색 구조와 이를 구현하는 하드웨어 구조를 제안한다. 제안된 구조는 [1]에서 제안된 binary tree의 문제점들을 해결하는 구조로서 작은 수의 엔트리를 갖는 TCAM과 pipelined binary search를 적용하여 효율적인 하드웨어로 구현되었다. 구현된 하드웨어 구조의 성능을 평가하여 본 결과, 약 30,000 여개의 entry를 갖는 MAE-WEST 라우터 데이타의 경우, 2,000여개의 엔트리를 갖는 TCAM과 총 245 Kbyte의 SRAM을 사용하여 한번의 메모리 access를 통하여 어드레스 검색이 가능한 것으로 평가되었다. 또한 제한된 방식은 큰 데이터베이스나 IPv6를 위해서도 확장이 용이하다.

Key Words : Binary Prefix Tree, Pipelined Binary Search, Longest Prefix Match, EnBiT

ABSTRACT

Efficient hardware implementation of address lookup is one of the most important design issues of internet routers. Address lookup significantly impacts router performance since routers need to process tens-to-hundred millions of packets per second in real time. In this paper, we propose a practical IP address lookup structure based on the binary tree of prefixes of different lengths. The proposed structure produces multiple balanced trees, and hence it solves the issues due to the unbalanced binary prefix tree of the existing scheme. The proposed structure is implemented using pipelined binary search combined with a small size TCAM. Performance evaluation results show that the proposed architecture requires a 2000-entry TCAM and total 245 kbyte SRAMs to store about 30,000 prefix samples from MAE-WEST router, and an address lookup is achieved by a single memory access. The proposed scheme scales very well with both of large databases and longer addresses as in IPv6.

* 이화여자대학교 정보통신학과 Network SoC Design 연구실 (hlim@ewha.ac.kr)
논문번호 : 030388-0903, 접수일자 : 2003년 9월 5일
※본 연구는 한국학술진흥재단 BK21 지원으로 수행되었습니다.

I. Introduction

인터넷 라우터에서 실시간으로 수행되어야 하는 주요 기능은 입력된 패킷을 최종 목적지를 향하여 전달 (forward) 하는 것이다. 이를 위하여 모든 라우터는 전달 테이블 (forwarding table)을 갖으며, 이 전달 테이블로부터 목적지 어드레스 (destination address)에 따르는 출력 포트와 다음 홉 주소들에 대한 정보들을 얻어낸다. 패킷이 입력되면 라우터는 입력된 패킷의 목적지 어드레스의 네트워크 파트를 key로 사용하여 전달 테이블을 참조하는 일을 수행하는 데, 이를 어드레스 검색이라고 하고 이때 사용되는 네트워크 파트를 prefix라고 한다.

클래스를 갖는 어드레싱 방식(Classful Addressing Scheme)에서는 IP 어드레스의 prefix를 8 bit, 16 bit, 혹은 24 bit으로 고정 시켜 놓고 사용하였다. 이 방식은 두 가지 문제점을 갖는데, 첫 번째는 prefix가 고정되어 있으므로 IP 어드레스가 낭비되는 것이고, 둘째는 네트워크의 종류가 증가함에 따라 라우터의 전달 테이블 크기가 기하급수적으로 증가한다는 것이다.

이러한 문제점을 해결하기 위하여 클래스를 갖지 않는 어드레싱 방식 (CIDR: Classless Interdomain Routing)이 나오게 되었는데 이 방식에서는 prefix의 길이를 고정시켜 놓지 않음으로 IP 어드레스의 낭비를 막을 수 있다. 또한 어드레스 aggregation을 가능하게 하여 라우터에서 전달 테이블의 크기가 빠르게 증가하는 것을 방지할 수 있었다. CIDR 방식의 단점은 라우터에서 longest prefix matching을 수행하여야 한다는 것이다. 이는 입력된 패킷은 자신의 최종 네트워크의 prefix 길이에 대한 정보를 포함하고 있지 않으므로, 전달 테이블에 나와 있는 prefix들 중에서 입력된 패킷의 어드레스와 가장 길게 일치하는 엔트리를 찾아야 하는 것이다. 다시 말하면 그동안 exact matching을 위해 사용된 여러 가지 어드레스 검색 방식들을 longest prefix matching 방식을 위해서는 사용할 수 없게 되었다.

인터넷 라우터에 연결되어 있는 link의 속도는 10Gbps 이상의 속도까지 증가할 것으로 전망되며, prefix의 종류도 다양하여져서 백본 라우터의 전달 테이블 엔트리의 수는 100K이상으로 증가할 것으로 보인다. 이러한 환경에서 longest prefix matching에 의한 어드레스 검색은 오늘날의 라우터의 성능을 결정짓는 주요 bottleneck으로 작용하게 되어 효율적인 IP 어드레스 검색 알고리듬과 구조에 대한 연구가 활발하게 진행되어 왔다.

본 논문에서는 [1]에서 제안된 prefix들의 binary tree 구조는 IP 어드레스 검색을 위한 매우 효율적인 데이타 구조이나 구성된 tree가 unbalanced 되어진데서 오는 여러 문제점이 있음을 발견하고, balanced tree를 구성할 수 있는 개선된 데이타 구조와 이를 구현하는 하드웨어 구조를 제안한다.

본 논문의 구성은 다음과 같다. 먼저 II 장에서는 기존에 연구되어진 어드레스 검색구조에 관하여 정리한다. III 장에서는 [1]에서 제안하는 Binary Tree 데이타 구조를 요약하여 설명한다. IV 장에서는 [1]에서 제안된 데이타 구조의 단점을 개선하는 EnBiT (Enhanced Binary Tree) 구조를 제안한 후 V 장에서는 제안하는 데이타 구조를 구현하는 하드웨어 구조를 보여준다. VI 장에서는 MAE-WEST 라우터의 실제 데이타를 사용하여 제안하는 구조의 성능을 평가하였으며, 타 구조와의 성능을 비교하였다. VII 장에서 간단한 결론을 맺는다.

II. Previous Work

어드레스 검색은 접근 방식에 따라 크게 몇 가지로 구분하여 볼 수 있는데, 그 첫 번째가 trie를 사용하는 방식이다. Trie는 tree에 기초한 데이타 구조로써, prefix들 간의 관계를 알기 쉽게 표현한 가장 대표적인 데이타 구조라 할 수 있다. 이 구조에서는 모든 prefix는 trie에서의 하나의 노드에 위치하고 root로부터 출발하는 path로 정의된다. 그림 1에서는 몇 개의 prefix들을 표현하는 간단한 trie를 예로 보여 준다 [2]. Trie 구조를 효율적으로 메모리에 저장하는 여러 가지 방식과 이에 따르는 다양한 IP 어드레스 검색 방식들이 연구되어 왔다 [3]-[7].

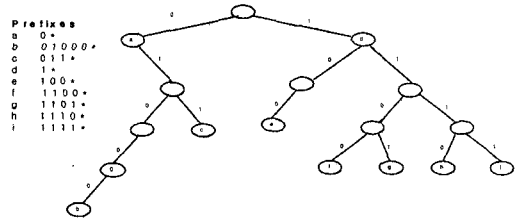

그림 1 프리픽스 트리의 예

Trie에 기초한 방식은 prefix에 해당되지 않는 internal 노드들을 저장하는데서 오는 메모리의 낭비와 W를 trie의 높이라 할 때 O(W) 번의 메모리 검색을 수행하여야 하는 문제점을 갖는다.

두번째는 같은 길이를 갖는 prefix 들끼리의 해싱 (Hashing)을 사용하는 방식이다. 해싱은 exact matching 방식을 사용하는 Layer 2 주소 검색을 위해 널리 사용되어 온 방식이다.

[8],[9]에서는 binary search와 hashing을 결합하여 사용하는 방식을 제안하고 있다. 그러나 이 방식의 경우, 순수한 binary search가 적용되지 않는 문제점을 안고 있다. 다시 말하면 hashing에 의해 검색된 entry가 prefix를 포함하고 있지 않더라도, IP 주소의 longest prefix match의 특성 때문에 더 긴 prefix가 존재 할 수 있으므로, 더 긴 prefix가 존재함을 알려주기 위해 marker를 저장하여야 하는 것이다. 이때 수많은 marker를 미리 계산하고 저장하는 과정에 overhead가 따른다. 또한 이 구조는 특정 prefix distribution에 대하여 빠른 시간 내에 perfect hashing 함수를 찾을 수 있음을 가정하고 있기 때문에 실용적이라고 볼 수 없다.

[10]에서는 prefix 길이 별로 어드레스 검색 테이블을 만들고, prefix 길이에 따라 별도의 해싱 함수를 사용하여 모든 prefix 길이에 대하여 병렬 검색을 수행하는 방식을 제안하고 있다. Prefix 길이 별 별도의 해싱 테이블로부터 match되어 나온 prefix 중 가장 길게 match된 prefix를 찾는 방식이다. 이 논문은 perfect hashing 함수를 가정함 없이, EXOR로 구현된 hashing 함수를 사용하였고 hashing에서 일어나는 충돌현상은 sub-table을 두어 해결한 구조로서 실용적인 구조라고 할 수 있다. 그러나 주 테이블에서 충돌현상이 발생한 경우 보조 테이블에서 binary 검색을 수행함으로 최대 메모리 접근 횟수가 큰 단점이 있다.

[11]에서는 hashing에서의 충돌 현상 (collision)을 해결하기 위하여, 여러 개의 hashing 테이블을 사용하여 현재 충돌 횟수가 적은 쪽의 entry에 새로운 prefix를 저장하는 방식을 제안하고 있다. 그러나 이러한 데이타 구조를 구현하는 하드웨어 구조에 관하여서는 연구된 바가 없다. 앞서 언급된 hashing을 사용한 모든 어드레스 검색 구조들은 hashing구조의 본질적인 문제인 메모리 접근 횟수가 가변적이며, 비어 있는 엔트리에 대한 메모리 낭비는 해결하지 못한 단점을 지닌다.

세번째로, 현재 실제 스위치나 라우터의 구현에 많이 사용되어지고 있는 기술로서 TCAM (Ternary Content Addressable Memory)을 사용하는 방식이다 [12]. 이 방식은 입력된 주소와 TCAM에 저장된 모든 prefix들을 동시에 직접 비교하여 매우 빠른 어드레스 검색이 가능하다. 그러나 TCAM 관련 기술은 라우터에서 사용하는 prefix 수의 증가만큼 빠르게 발전하고 있지 못하다. 다시 말하면 같은 공간 크기를 차지하는 RAM에 비하여 TCAM에 저장할 수 있는 prefix의 갯수는 매우 적고 값도 비쌀 뿐 아니라, 전력 소모도 매우 커서 수 만개의 prefix를 저장하는 TCAM을 어드레스 검색을 수행하기 위해 설계된 칩 내부에 내장하기 어려운 문제점을 지닌다.

마지막으로 가장 최근에 발표되어진 방식으로, 길이가 다른 prefix들간의 binary search를 적용하는 것이다 [1],[13],[14]. Binary search 방식 또한 해싱과 마찬가지로 exact matching 검색을 사용하는 Layer 2 어드레스 검색을 위하여 사용된다. 그러나 길이가 다른 prefix들간의 비교를 수행하여야 하는 Layer 3 IP 어드레스 검색을 위해서는 사용되지 못하여왔다. [1]에서는 서로 길이가 다른 두개의 prefix들간의 크기를 정의하고, 하나의 prefix가 다른 여러 개의 prefix들을 포함하고 있을 경우의 enclosure를 정의하여 IP 어드레스 검색을 위한 binary search를 가능하게 하였다. 이러한 방식으로 tree를 구성하였을 경우 모든 노드에 prefix를 할당하므로 메모리의 낭비가 없이 효율적으로 prefix들을 저장할 수 있는 장점을 지닌다. 그러나 N개의 라우트를 갖는 prefix들의 binary search는 $O(\log_2 N)$ 의 메모리 검색을 필요로 하는 문제점을 지닌다. [13]에서는 [1]에서 제안된 데이타 구조를 구현하여 어드레스 검색을 수행하는 하드웨어 구조를 제안하였으나, 이는 제안된 데이타 구조가 갖는 특징을 간과한 구조로서 효율적이지 못하다고 할 수 있다.

본 논문은 [1]에서 소개된 binary tree 데이타 구조는 prefix들간의 관계를 효과적으로 표현하나 tree가 balanced 되어 있지 않아 구현이 어려운 데서 착안하였다. 본 논문에서는 [1]의 binary tree 구조를 개선하여 전체 tree가 balance 되는 구조를 제안하여 unbalanced tree에서 오는 모든 문제점을 해결하였다. 또한 제안하는 데이타 구조를 구현하는 매우 효율적인 하드웨어 pipeline 구조를 제안한다.

Ⅲ. Binary Prefix Tree Structure

본 논문에서는 [1]에서 제안된 데이타 구조를 Binary Prefix Tree(BPT) 구조라 부르기로 한다. BPT 구조는 prefix들의 크기에 따른 정렬로부터 출발한다. 길이가 다른 prefix들 간의 binary 검색을 위하여 BPT는 다음과 같은 정의를 사용하였다.

정의 1 (크기 비교에 관한 정의): 두개의 prefix A = $a_1a_2.....a_n$ 과 B = $b_1b_2.....b_m$을 가정 할 때, 만약 n=m이면, A와 B의 numerical 값이 비교되어진다.

1. 만약 n!=m (n<m을 가정) 이면, substring $a_1a_2.....a_n$ 과 $b_1b_2.....b_n$ 이 비교되어 더 큰 numerical값을 가지는 prefix가 더 큰 것으로 정의한다. 예를 들어 A=1001, B=110000인 경우 B는 A보다 크다.
2. 만약 두개의 substring이 같은 경우, (n+1) 번째 bit이 1이면 B>A, 아니면 B<A 이다. 예를 들어 A=1001, B=100110인 경우 B는 A보다 크고 A=1001, B=100100이면 A가 B보다 크다.

정의 2 (match에 관한 정의): 두개의 prefix A = $a_1a_2.....a_n$ 과 B = $b_1b_2.....b_m$을 가정 할 때, n=m이고 두개의 스트링이 같거나, n<m일 경우 $a_1a_2.....a_n$ 와 B의 substring $b_1b_2.....b_n$이 같으면 match 한다. 그렇지 않으면 A와 B는 match하지 않는다. 예를 들어 A=1001, B=100100인 경우 A와 B는 매치한다.

정의 3 (disjoint 에 관한 정의): 두개의 prefix A와 B가 어느 쪽도 다른 쪽의 prefix가 아니면 A와 B는 disjoint 하다. 예를 들어 A=1001, B=111인 경우 A와 B는 disjoint이다.

정의 4 (enclosure에 관한 정의): prefix A를 prefix로 갖는 다른 prefix가 하나라도 존재하면 A는 enclosure이다. 예를 들어 A=1001, B=100100인 경우 A는 B의 enclosure이다.

BPT를 구성하는 과정은 다음과 같은 단계를 거친다. 먼저 정의 1을 사용하여 모든 prefix들을 크기 별로 나열한 리스트를 만든다. 그림 2에는 이 과정을 수행하는 알고리듬을 보였다. 먼저 모든 prefix 들을 서로 비교하여, 어떤 prefix가 다른 prefix의 enclosure(정의 4)인 경우 enclosure의 bag 속에 prefix를 넣는 과정을 수행한다. 예를 들면, 위의 정의 4에서 보이는 예의 경우 B는 A의 bag에 들어가게 된다. 이 과정을 수행하고 나면 서로 disjoint (정의 3)한 prefix들만 남게 되고, 이 prefix들을 크기별로 정렬하는 과정을 수행한다. 그림 2의 알고리듬은 worst case에 $O(N^2)$의 성능을 갖는다.

다음은 BPT를 만드는 과정이다. 그림 2의 알고리듬에 의해 만들어진 리스트는 recursive하게 반으로 나누어져 binary tree가 생성된다. 여기서 한가지 유의해야 할 사항은 만약 나누어지는 점의 prefix가 정의 4에서의 enclosure인 경우, enclosure의 bag속에 포함된 모든 데이타들을 이 enclosure를 root로 하는 sub-tree에 포함시켜야 한다는 것이다. 이는 prefix란 하나의 point를 가리키는 것이 아니라, 그 prefix로 시작하는 range를 모두 포함하는 것으로서, binary search는 상위 레벨이 먼저 검색되기 때문에 enclosure를 상위 레벨에 두어 먼저 검색하기 위함이다. 그림 3에 BPT를 만드는 알고리듬을 보였다. 그림 3의 알고리듬은 $O(N^2\log_2N)$의 성능을 갖는다.

그림 4에는 표 1에서 보인 prefix 들의 BPT 구조를 예로 보였는데, 그림 4에서 보듯이 enclosure prefix 10*를 root로 하는 sub-tree에는 10*를 prefix로 갖는 모든 prefix 들이 포함되어 있음을 알 수 있다. 그림 4에서 색칠이 되어 있는 노드는 enclosure prefix임을 나타내고, 색칠이 되어 있지 않는 노드는 disjoint prefix임을 나타낸다. 그림 4

```
Sort Procedure
/*List contains all initial and sorted prefixes. */
Sort(List)
/*First find enclosures */
for all i in List do;
    compare i with all j in List where j!= i;
    if i matches j then
          if i is shorter than j, then /*i is enclosure of j*/
            put j in i's bag.
            delete j from List.
          else
            put i in j's bag.
            delete i from List.
    end compare;
end for;
/*Now sort them */
for all i in List do;
          compare i with all j in List where j!= i;
          if j<i, then
                exchange i with j.
          end compare;
 end for;
end Sort;
```

그림 2 프리픽스의 크기별 정렬 알고리듬

```
Building Tree Procedure
BuildTree(List)
    if List is empty, return.
    Sort(List)
    let m be the median of List
    root ← m;
    let leftList and rightList contain all elements in the left and right of m.
    if m is an enclosure, then,
      distribute elements in m's bag into leftList and rightList.
    leftChild(root)← BuildTree(leftList);
    rightChild(root)← BuildTree(rightList);
    return address of root.
end BuildTree;
```

그림 3 바이너리 프리픽스 트리를 구성하는 알고리듬

에서 볼 수 있듯이 34 개의 prefix들의 binary tree가 10개의 레벨을 갖는 unbalanced tree로 구성되었다.

이런 과정을 거쳐 만들어진 BPT에서의 어드레스 검색 알고리듬은 그림 5에 보여준 알고리듬을 사용한다. 그림 5에 보여준 검색 알고리듬에서는 search procedure를 recursive하게 부름으로서, 하위 레벨에서 match가 없는 경우에만 상위 레벨에서의 match되는 prefix를 찾으므로, 하위 레벨에서 match된 prefix가 longest prefix로 선정된다. 이를 하드웨어로 설계 할 때에는 상위 레벨에서 match된 prefix를 하위 레벨에서 match된 prefix로 대치하는 방식으로 설계한다. 이는 앞서 말한 바와 같이 enclosure를 상위 레벨에 두고 하위 레벨에서 matching 노드가 생기면 이는 enclosure보다 더 긴 prefix이므로 enclosure를 대치함으로써 IP 어드레스의 longest prefix matching을 구현하기 위함이다. 앞서 설명한 바와 같이, BPT에서의 어드레스 검색은 $O(\log_2 N)$의 성능을 갖는다.

BPT에서의 update는 두 경우로 나누어 설명할 수 있다. 추가되는 prefix가 정의 3에서 보여준 disjoint한 prefix라면, tree의 leaf에 추가할 수 있으므로 앞서 설명한 어드레스 검색과 같은 성능을

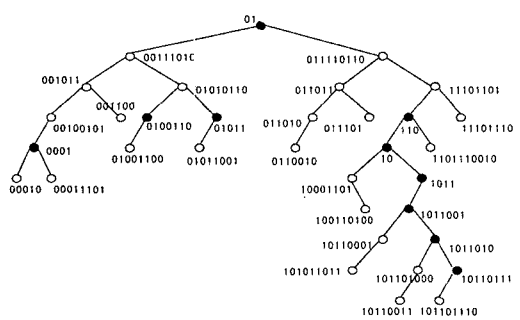

그림 4 표 1 의 프리픽스들의 바이너리 프리픽스 트리

표1 프리픽스들의 예

P_1	10	P_{18}	1101110010
P_2	01	P_{19}	10001101
P_3	110	P_{20}	11101101
P_4	1011	P_{21}	01010110
P_5	0001	P_{22}	00100101
P_6	01011	P_{23}	100110100
P_7	00010	P_{24}	101011011
P_8	001100	P_{25}	11101110
P_9	1011001	P_{26}	10110111
P_{10}	1011010	P_{27}	011010
P_{11}	0100110	P_{28}	011011
P_{12}	01001100	P_{29}	011101
P_{13}	10110011	P_{30}	0110010
P_{14}	10110001	P_{31}	101101000
P_{15}	01011001	P_{32}	101101110
P_{16}	001011	P_{33}	00011101
P_{17}	00111010	P_{34}	011110110

갖는다. 그러나 만약 추가되는 prefix가 BPT에 이미 존재하는 다른 prefix의 enclosure인 경우, 추가되어지는 enclosure는 기존 prefix를 대치하여야 하고, 새로 추가된 enclosure를 root로 하는 sub-tree를 다시 구성하여야 하는 문제점이 있다. BPT에서의 테이블 update 알고리듬은 그림 6에 보였다.

Ⅳ. EnBiT (Enhanced Binary Tree)

Ⅳ장에서는 [1]에서 제안된 BPT 구조를 수정하여 문제점을 개선할 수 데이타 구조를 제안한다. 제안하는 구조를 EnBiT(Enhanced Binary Tree)이라 부르기로 한다. Ⅲ장에서 설명한 BPT는 특정 prefix를 enclosure로 하는 다른 prefix들을 그 enclosure를 root로 하는 sub-tree에 포함 시켜야 함을 언급하였다. 이는 longest prefix match를 수행하여야 하는 IP 어드레스 검색의 특성상, 입력된 목적지 IP 어드레스가 어떤 enclosure prefix와 match하였다 하더라도 더 길게 match하는 prefix를 찾아야 하기 때문이다. 이러한 제약 때문에 앞의 그림 4에서 보였듯이 unbalanced 한 tree가 생성되며, 이러한 unbalanced 한 tree는 하드웨어로 구현되기에 어려움을 지닌다.

EnBiT은 tree의 구성 요소인 prefix들이 서로 disjoint한 관계를 갖는다면 balanced tree를 구성할 수 있다는 점에서 착안하여 BPT의

```
Search Procedure
/*tree points to the root and addr is an IP address. */
Search(tree,addr)
      if tree=NIL,return NULL;
      if (addr<tree(root)),
        prefix←Search(leftChild(tree),addr);
      else
        prefix←Search(rightChild(tree),addr).
      if addr matches tree(root) and prefix is NULL,
        prefix←tree(Node).
      return prefix;
end Search;
```

그림 5 바이너리 프리픽스 트리의 어드레스 검색 알고리듬

unbalanced 성질을 개선하기 위해 제안되었다. 이는 enclosure들의 sub-tree가 main tree에서 분리하여 독립된 tree로서 존재하면 가능한 것으로서 EnBiT에서는 enclosure들의 sub-tree를 main tree와 분리하여 독립된 sub-tree로 만든다. 즉 main tree는 disjoint prefix들 만으로 구성되고, 모든 enclosure들은 자신을 enclosure로 하는 prefix들의 sub-tree로 가는 포인터를 가지며, 모든 sub-tree의 prefix들 또한 서로 disjoint한 관계를 갖는다.

그림 7에 EnBiT을 위해 수정된 Sort 알고리듬을 보였다. Main tree와 모든 enclosure의 sub-tree들을 recursive하게 sorting 하는 알고리듬이다. Sorting 결과 main tree 리스트와 enclosure set 에 속한 여러 개의 sub-list가 생성된다. EnBiT에서 tree를 만들기 위해 수정된 알고리듬은 그림 8에 보였다. 그림 8의 알고리듬은

```
Insertion Procedure
/*tree points to the root and prefix is an IP prefix. */
 Insertion(tree,prefix)
      if tree=NIL,then,
            node←AllocateNode();
            node←prefix;
            make tree parent of node;
            return;
      if prefix is enclosure of tree(root) then;
            replace tree(root) with prefix;
            Insertion(tree,tree(root);
            if prefix < tree(root), then;
                  Move(leftChild(tree),prefix);
            else
                  Move(rightChild(tree),prefix);
            return;
      if prefix < tree(root), then;
            Insertion(leftChild(tree),prefix).
      else
            Insertion(rightChild(tree),prefix).
end Insertion;
```

그림 6 바이너리 프리픽스 트리의 업데이트 알고리듬

```
Sort Procedure
/*List contains all initial and sorted prefixes. */
Sort(List)
/*First find enclosures */
for all i in List do;
      compare i with all j in List where j!= i;
      if i matches j then
            if i is shorter than j, then /*i is enclosure of j*/
              put j in i's bag.
              delete j from List.
            else
              put i in j's bag.
              delete i from List.
      end compare;
end for;

/* Remove enclosure from List and Sort bag */
for all i in List do
      if i's bag is empty, return
      add i to enclosure set
      Sort (i's bag);
      delete i from List;
end for

/*Now sort them */
for all i in List do;
            compare i with all j in List where j!= i;
            if j<i, then
                  exchange i with j.
            end compare;
 end for;
end Sort;
```

그림 7 EnBiT의 Sort 알고리듬

main tree의 리스트와 enclosure 집합에 들어있는 모든 enclosure의 리스트들을 위해 반복적으로 수행되어야 한다. 그림 9는 그림4와 같은 prefix sample을 이용한 EnBiT의 예를 보여준다. 그림9에서 검은색 노드는 enclosure set에 속한 prefix들을 나타내며, 입력된 prefix가 enclosure에 속한 prefix와 match하면 match된 enclosure prefix가 가리키는 sub-tree로 이동하여 검색을 진행한다. 입력된 prefix가 현재 노드에 저장된 prefix보다 클 때는 g로 표시된 노드로 이동하고, 작을 때는 l로 표시된 노드로 이동하여 검색을 진행한다.

EnBiT은 main tree와 여러 개의 sub-tree로 구성되므로 어떤 tree에서 binary search를 진행할 것인지 조기에 결정하여, 보다 작은 검색 공간 내에서 효율적인 binary search를 가능하게 한다. EnBiT에서의 어드레스 검색은 입력된 어드레스와 가장 길게 match하는 enclosure prefix를 찾는 것으로부터 시작한다. 입력된 어드레스가 어떤 enclosure 와도 match 하지 않는 경우에는 disjoint한 prefix 들로만 구성된 main tree로 이동하여 binary search를 진행하고, tree에서 매치하는 엔트리를 만나면 그 엔트리의 fwd RAM 포인터를 출력한 후 조기에 검색을 종료한다. 이는

```
Building Tree Procedure
BuildTree(List)
      if List is empty, return.
      Sort(List)
      let m be the median of List
      root <= m;
      let leftList and rightList contain all elements
      in the left and right of m.
      leftChild(root)  <= BuildTree(leftList);
      rightChild(root) <= BuildTree(rightList);
      return address of root.
end BuildTree;
```

그림 8 EnBiT에서의 트리 구성 알고리듬

EnBiT의 tree내의 모든 prefix들은 서로 disjoint 한 관계에 있으므로, 하나의 엔트리와 매치 하였다면 다른 엔트리와는 매치하지 않기 때문이다. Main tree 전체를 검색하였으나 매치되는 엔트리가 없는 경우에는 default fwd RAM 포인터를 출력한다.

입력된 어드레스가 enclosure들과 match하는 경우에는 가장 길게 match한 enclosure의 fwd RAM 포인터를 기억한 후, 선택된 enclosure의 sub-tree에서 binary search를 이용한 어드레스 검색을 진행한다. Sub-tree에서 새로 match되는 prefix가 있으면 기억한 fwd RAM 포인터를 새로

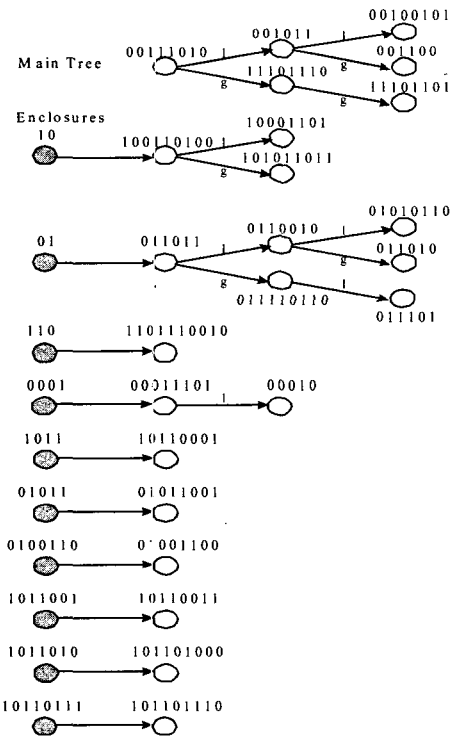

그림 9 그림 4와 동일한 prefix sample을 이용하여 구성한 EnBiT

match된 fwd RAM 포인터로 대치하고 검색을 종료한다. Sub-tree 전체를 검색하는 동안에 새로 match 되는 엔트리가 없다면, 현재 기억하고 있는 fwd RAM 포인터를 출력한 후 검색을 종료한다.

EnBiT 구조는 prefix update가 용이하다. EnBiT 에서 특정 prefix를 제거하기를 원하는 경우에는, 그 prefix의 fwd RAM 포인터가 무효함을 표시한다. 이런 방식으로 어드레스 테이블을 update 하는 경우 무효한 엔트리가 생길 것이나, 어드레스 검색에는 영향을 미치지 않는다. 어드레스 검색 시에는 무효한 엔트리의 prefix 값은 비교를 위하여 사용되나 fwd RAM 포인터는 기억되지 않는다.

그림 10은 EnBiT에서 새로운 prefix를 추가하는 경우에 대한 flow chart를 보여주고 있다. 첫 번째는 추가하고자 하는 새로운 prefix가 disjoint 한 경우이다. 이때는 main tree의 leaf로 저장하면 된다. 두 번째 경우는 새로운 prefix가 enclosure set에 있는 prefix의 enclosure인 경우이다. 이 경우 새로운 prefix를 enclosure set에 추가하면 된다. 세번째 경우는 새로운 prefix가 main tree나 sub-tree에 있는 prefix의 enclosure인 경우이다. 이 경우 검색된 prefix는 tree에서 제거되고 새로운 tree를 구성하게 된다. 새로운 prefix는 enclosure set에 추가되고 새로 구성한 tree를 기억하게 된다. 네 번째로는 새로운 prefix가 tree에 존재하는 prefix를 enclosure로 갖는 경우이다. 이때는 선택된 prefix는 enclosure set에 추가되고 새로운 prefix가 저장된 노드의 포인터를 기억한다. 새로운 prefix는 sub-tree의 leaf에 저장된다. 그림 10의 flow chart에서 나타낸 바와 같이 EnBiT에서는 incremental update가 가능함을 알 수 있다.

앞서 설명한 바와 같이 EnBiT 는 disjoint 한 prefix들로만 구성된 작은 높이의 여러 개의 balanced tree를 구성하는 데이타 구조로서 검색과 update 면에서 모두 우수한 성능을 갖는다.

V. Hardware Architectures

Binary Search 검색 알고리듬은 여러 가지 방식으로 병렬 처리가 가능하다. [13]에서는 BPT를 만들고, 어드레스를 검색하는 하드웨어를 제안하였다. [13]에서 제안된 방식은 메모리 접근 횟수를 줄이기 위하여 Cache의 역할을 하는 CAM을 두었다. 그러나 [13]에서 언급되어진 바와 같이 라우터에서의

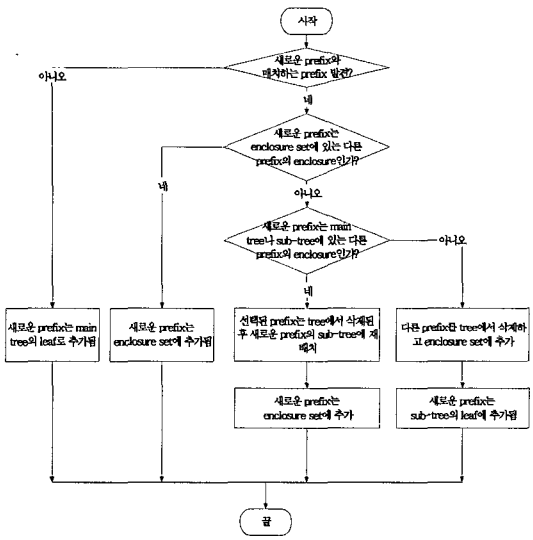

그림 10 EnBiT에서의 새로운 prefix 추가에 대한 flow chart

Cache Hit 비율은 50% 이하로서 효과적이라 할 수 없다. 또한 M-way Prefix Tree를 이용하여 메모리 엑세스 횟수를 더욱 줄이고자 시도하였는데 이에 따라 메모리 사용 효율이 현저히 저하되었다. 예를 들어 128K 개의 prefix들을 8-way Prefix Tree에 저장하려 할 때, 약 35%의 노드가 비어있음을 알 수 있다 [13]. 더군다나 leaf 노드가 전체 노드의 70% 이상이 되고, branching factor의 증가에 따라, 요구되는 ASIC의 입출력 pin의 수 또한 많아지는 단점이 있다. 결론적으로 [13]에서 제안한 하드웨어 구조는 BPT 구조의 최대 장점인 비어있는 노드가 없어 효율적으로 메모리에 저장될 수 있는 특성을 활용하지 못 하였을 뿐 아니라, binary search가 갖는 pipeline에 적합한 본질적인 성질을 간과한 구조로서 우수한 하드웨어 구조라 하기 어려울 것이다.

본 장에서는 binary search의 본질적인 성질을 이용하는 pipelining 방식에 기초한 하드웨어를 제안하고자 한다. 먼저 그림 4에 나타난 BPT를 예로 하여 본 논문에서 제안하는 pipeline 구조를 설명하고자 한다. 본 논문에서 제안하는 pipeline 구조는 binary search의 성질을 그대로 이용하는 것으로서, 트리를 레벨별로 가로로 나눈 뒤 같은 레벨에 속하는 모든 노드들을 묶어 하나의 메모리를 사용하여 저장하고 각 레벨별로 메모리 엑세스를 pipelining 하는 것이다. 각 엔트리에는 자신의 child로 가는 포인터들을 기억한다. 그림 11은 그림

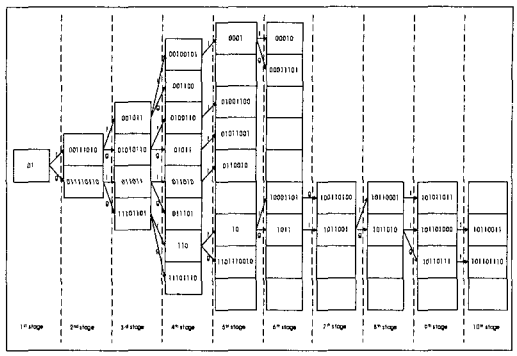

그림 11 바이너리 프리픽스 트리의 Pipeline 구조

4에 보인 트리에 본 논문에서 제안하는 pipeline 구조를 적용하였을 때를 보여준다. 입력된 패킷의 어드레스가 노드에 저장된 prefix보다 작은 경우는 그림의 l로 표현된 path를 따라 가고, 큰 경우에는 그림의 g로 표현된 path를 따라 검색을 진행한다.

앞서 EnBiT 구조에서의 어드레스 검색은 enclosure들 중 가장 길게 일치하는 enclosure를 검색하는데서 출발함을 언급하였다. 다음 장의 실험 결과에서 보였듯이 prefix distribution에서의 enclosure의 갯수는 7% 미만으로서 적은 수의 엔트리를 갖는 TCAM을 사용함으로 구현이 가능하다. 그림 12에 제안하는 pipeline 구조를 적용하여 그림 9의 EnBiT을 구현하는 하드웨어를 보였다. 그림 12에서는 굵은 선으로 표시된 것이 하나의 tree를 나타내며 하단의 트리가 main tree이다. 어드레스 검색 과정에서 input이 enclosure와 일치하면 m이 가리키는 sub-tree를 따라 검색을 계속하게 된다. 만약 enclosure와 매치하지 않으면 main tree를 가리키는 default entry와 매치하게 된다.

TCAM과 각 tree의 노드는 prefix, prefix 길이와 fwd RAM 포인터를 저장하고 있다. 추가적으로

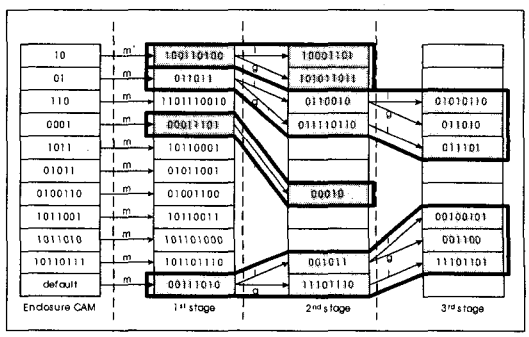

그림 12 제안하는 첫번째 하드웨어 구조

TCAM의 엔트리에는 자신의 sub-tree로 가는 포인터가 tree의 노드는 children으로 가는 포인터가 저장된다. 그림 12에서 보인 레벨별로 별도의 메모리를 사용하여 노드들을 저장하는 경우, 각 레벨에서의 메모리 검색은 pipelining이 가능하다. 다음 장에서 보이겠지만 60% 이상의 prefix가 main tree에 포함되므로 pipeline 단계의 수는 main tree의 높이에 의하여 결정된다. 그림12의 어드레스 검색은 가장 길게 매치하는 enclosure에서 시작하여 main tree나 sub-tree중 매치하는 엔트리에서 끝나게 된다. 그림 12에서 볼 수 있듯이 pipeline stage는 3으로 그림 11의 pipeline stage와 비교하였을 때 현저히 줄어든 것을 알 수 있다. EnBiT에서의 메모리 엑세스는 input stream에 따라 pipeline되어 결과적으로 어드레스 검색은 한번의 TCAM access나 한번의 메모리 엑세스로 가능하게 된다.

Enclosure의 수가 많은 경우를 위해 두 번째 구조를 제안하였다. 200K개 이상의 prefix를 가진 forwarding table이라고 하였을 때 14K-entry를 가진 TCAM은 현재의 기술로는 칩에 내장되기 어렵다. 게다가 IPv6의 경우 IPv4에 비해 어드레스의 길이가 길어지기 때문에 더 많은 enclosure를 가지게 될 것이다. 두 번째 구조에는 TCAM에 첫 단계에서 생성된 enclosure들만을 저장하는 방법을 제안한다. 그림 13에서 볼 수 있듯이 두 번째나 세 번째 레벨의 enclosure를 메모리 pipelining에 포함키고 m이 가리키는 노드에서 새로운 sub-tree가 시작된다.

예를 들면, 그림 13에서는 enclosure 10*이 가리키는 첫 번째 level 트리와, enclosure 1011*이 가리키는 두 번째 level tree, 그리고 enclosure 1011010*, enclosure 1011001* 과 enclosure

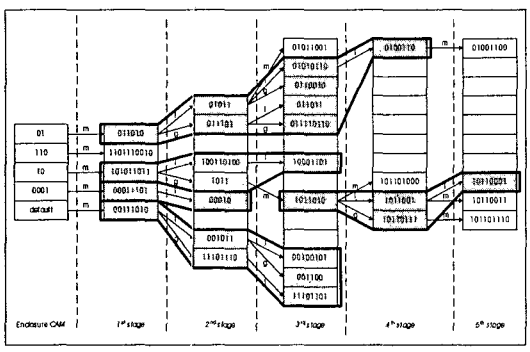

그림 13 제안하는 두번째 하드웨어 pipeline 구조

10110111*이 가리키는 3번째 level tree 가 존재하여 최대 3개 level의 sub-tree가 존재함을 알 수 있다.

그림 13의 구조에서의 어드레스 검색은 그림 12와 마찬가지로 가장 길게 매치하는 enclosure에서 시작하여 disjoint한 prefix에서 끝나게 된다. 즉 일치하는 엔트리가 있고, 그 엔트리가 새로운 sub-tree를 가리키는 포인터를 가지면, 새로운 sub-tree에서 검색을 진행한다. 이 과정은, 새로운 sub-tree를 포함하지 않는 노드, 즉 disjoint한 노드를 만날 때까지 계속된다.

VI. Simulation and Performance Evaluation

제안하는 하드웨어 구조를 C 언어와 Verilog 언어를 사용하여 2002년 3월 15일자 MAE-WEST 라우터에서의 실제 데이타를 사용하여 성능을 실험하였다. 먼저 MAE-WEST 데이타를 사용하여 [1]에서 제안하는 방식으로 하나의 BPT를 구성하여 보았을 때, 높이가 32 인 거대한 tree가 생성되었다. 이는 [1] 구조의 문제점인 unbalanced tree 구조에서 기인하는 것으로서, pipeline 단계의 수가 deterministic하지 않아 구현이 어렵다고 할 것이다.

제안하는 구조의 성능을 평가하기 위해 먼저 MAE-WEST 데이타를 분석한 결과 표 2와 같은 성질을 가짐을 알 수 있었다. 표 2에서 보이듯이 MAE-WEST 데이타의 경우 최대 5개 level의 sub-tree를 갖으며, 전체 prefix 중에 main tree에 포함될 disjoint한 prefix (첫번째 레벨에서의 disjoint한 prefix의 수)는 전체의 64.23%를 차지한다. Main tree에 들어가는 prefix 들은 높이가 15인 tree로 구성되며 15 단계를 갖는 pipeline 구조로 구현될 수 있다. 표 2로부터 첫 번째 제안한 구조와 두 번째 제안한 구조의 TCAM의 엔트리 수는 각각 1962개와 1599개인 것을 알 수 있다.

또한 시뮬레이션을 통해 EnBiT sub-tree의 최대 높이는 첫 번째 구조의 경우 9, 두 번째 구조의 경우 13으로 구해졌다. 두 경우 모두 main tree의 높이인 15안에 들어있음을 알 수 있었다. 즉 본 논문에서 제안하는 구조는 main tree에 들어가는 disjoint 한 노드들의 수를 산출해 내면, main tree는 balanced 한 구조를 가지므로 전체 pipeline 단계의 수를 산출해 낼 수 있다.

표 2 MAE-WEST 데이타에서의 프리픽스 분포

	Enclosure prefix 수	Disjoint prefix 수	전체 prefix 수
1st Level Tree	1,599	19,001	20,600
2nd Level Tree	312	7,538	7,850
3rd Level Tree	50	941	991
4th Level Tree	1	140	141
5th Level Tree	0	2	2
Total	1,962	27,622	29,584

제안하는 구조에서의 각 tree의 노드는 노드의 유효함을 나타내기 위해 1bit, prefix 길이를 표시하기 위해 5 bit, prefix 그 자체를 저장하기 위해 32 bit, fwd RAM pointer를 저장하기 위해 5bit, 그리고 children을 가리키는 2개의 14 bit 포인터, 도합 71 bit 이 필요하다. 따라서 27600개의 disjoint한 prefix를 저장하기 위해 245Kbyte의 SRAM이 필요한 것을 알 수 있다. tree의 leaf 노드에서는 children을 위한 포인터가 필요 없으므로 메모리 size는 좀 더 줄어들 것이다.

표 3에는 메모리 엑세스 횟수와 요구되어지는 메모리의 크기에 있어 제안하는 구조와 서론에서 언급한 타 구조를 비교하였다. 표 3에서 볼 수 있듯이 제안하는 구조는 메모리 엑세스 횟수와 요구되는 메모리 크기에 있어 월등히 우수한 구조라고 할 수 있다. 또한 제안된 구조의 모든 pipeline 단계는 모두 같은 구조를 가지고 있어 VLSI로 설계하기에 매우 좋은 구조라고 할 수 있다.

Ⅷ. Conclusions

본 논문에서는 [1]에서 제안된 BPT의 문제점을 개선하는 데이타 구조를 보이고, 이를 구현하는 효율적인 하드웨어 구조를 제안하였다. 제안된 구조는 BPT 구조가 갖는 최대 장점인 Tree 내에 빈 노드가 없다는 점, 그리고 binary search는 pipeline을 사용하여 구현되어 질 수 있는 점을 살려서 메모리를 효율적으로 사용하고, 최대 한번의 메모리 엑세스를 통하여 어드레스 검색을 수행할 수 있는 실용적이면서도 우수한 구조이다. 제안하는 구조는 또한

표 3 다른 방법들과의 성능 비교

Address Lookup Scheme	Number of Memory Accesses (Min/Max)	Forwarding Table Size
Huang's scheme [5]	1/3	450KB ~ 470KB
DIR-24-8 [6]	1/2	33MB
DIR-21-3-8 [6]	1/3	9MB
Pararell Hashing [10]	1/5	189 KB
Proposed Architecture	1/1	2000-entry TCAM 245KB memory

테이블 update에도 매우 우수한 성질을 갖으며, IPv6를 위해서도 쉽게 확장될 수 있는 장점을 지닌다. MAE-WEST를 통과한 실제 prefix의 분포를 가지고 실험한 결과, 전체 pipeline 단계가 15 인 구조가 생성되었고, 약 2000여개의 엔트리를 갖는 TCAM과 15개의 pipelined stage를 위해 전체 필요한 메모리의 크기는 245 Kbyte이다.

Acknowledgement

본 논문에서 사용된 실험을 위해 도움을 준 김옥하, 정주희 학생께 감사드립니다.

참 고 문 헌

[1] N.Yazdani and P.S.Min, "Fast and Scalable Schemes for the IP Address Lookup Problem", IEEE Conf. on High Performance Switching and Routing, pp 83-92, 2000.

[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] W. Chen and C. Tsai, "A Fast and Scalable IP Lookup Scheme for High-Speed Networks," Proceedings of IEEE ICON'99.

[4] D. Taylor, J. Rurner, J. Lockwood, T. Sproull, and D. Parlour, "Scalable IP Lookup for Internet Routers," IEEE Journal

of Selected Areas in Communications, vol. 21, No. 4, pp. 522-533, May 2003

[5] Nen-Fu Huang and Shi-Ming Zhao, "A Novel IP Routing Lookup Scheme and Hardware Architecture for Multigiga bit Switching Routers", IEEE journal on Selected Areas in Communication, Vol.17, No.6, pp.1093-1104,june 1999.

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

[7] V.Srinivasan and G.Varghese, "Fast address lookups using controlled prefix expansion," in Prac. ACM Sigmetrics'98 Conf., Madison, WI, pp.1-11, 1998

[8] M.Waldvogel, G.Varghese, J. Turner, and B.Plattner, "Scalable high speed IP routing lookups," in Prac. ACM SIGCOMM'97 Conf., Cannes, France, pp. 25-35, 1997

[9] M.Waldvogel, "Multi-Dimensional Prefix Matching Using Line Search", IEEE Conf. on Local Computer Networks, pp. 200-207,2000.

[10] Hyesook Lim, Ji-Hyun Seo and Yeo-Jin Jung," High Speed IP Address Lookup Architecture Using Hashing", 한국통신학회 논문지, 28권, 2B, p138-p143, 2003

[11] A.Broder and M.Miltzenmacher, "Using Multiple hash Functions to Improve IP Lookups", IEEE INFOCOM, pp. 1454-1463, 2001

[12] A.McAuley and P. Francis, "Fast routing lookup using CAM's," in Proc. IEEE INFOCOM, pp.1382-1391, 1993

[13] N.Yazdani and N.Salimi, "Performing IP Lookup on Very High Line Speed", EurAsia-ICT 2002: Information & Communication Technology: First EurAsian Conference Shiraz, Iran, Oct-29-31, 2002.

[14] N.Yazdani and P.S.Min, "Prefix Trees: New Efficient Data Structures for Matching Strings of Different Lengths", Database Engineering & Applications, International Synposiun on, pp 76-85, 2001

임 혜 숙(Hye-Sook Lim) 정회원

hlim@ewha.ac.kr

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 SoC설계, 통신관련 SoC 설계

이 보 미(Bo-Mi Lee) 준회원

shoutoi@ewha.ac.kr

2003년 2월 : 이화여자대학교 정보통신학과 학사

2003년 3월 ~ 현재 : 이화여자대학교 정보통신학과 석사과정

〈관심분야〉 Router나 switch등의 Network 관련 SoC설계, TCP/IP관련 하드웨어 설계

정 여 진(Yeo-Jin Jung) 준회원

surya@ewha.ac.kr

2003년 2월 : 이화여자대학교 정보통신학과 학사

2003년 3월 ~ 현재 : 이화여자대학교 정보통신학과 석사과정

〈관심분야〉 Router나 switch등의 Network 관련 SoC설계, TCP/IP관련 하드웨어 설계