論文2003-40SD-5-7

고속 IP Lookup을 위한 병렬적인 하이브리드 구조의 설계

(Design of Hybrid Parallel Architecture for Fast IP Lookups)

徐 大 植 *, 尹 晟 澈 *, 吳 在 錫 *, 姜 成 昊 *

((Dae-Sik Suh, Sung-Chul Yoon, Jae-Seuk Oh, and Sungho Kang)

요 약

네트워크 프로세서를 설계하거나, 라우터와 같은 장비를 구현할 때, 가장 성능을 좌우하는 부분이 IP lookup 동작이라 할 수 있다. 어드레스 체계가 간단해지면 IP lookup 동작이 단순화될 수 있어 성능이 좋아질 수 있지만, 네트워크 사용자의 증가로 인하여 효율적인 어드레스 관리가 필요하게 됨으로써 어드레스 체계는 복잡해질 수밖에 없는 상황이 되었다. 따라서 IPv4나 IPv6에서나 마찬가지로 어드레스 체계의 복잡성에 의해 IP lookup 동작이 어렵고 시간이 오래 걸리는 작업이 되는 것은 받아들일 수밖에 없는 현실이 되었다. 소프트웨어적으로나 하드웨어적으로 IP lookup 성능을 향상시킬 수 있는 방안들이 연구되어 왔지만, 아직까지 해결책이라고 단정지을 수 있을 만한 연구결과는 나오지 않고 있다. 소프트웨어적인 방법은 메모리 사용량을 줄일 수 있지만 IP lookup시 검색이 느리고, 하드웨어적인 방법은 빠르지만 하드웨어 오버헤드와 메모리 사용량이 크다는 문제가 있다. 이에 이 논문은 지금까지의 IP lookup 동작을 향상시키기 위한 연구들을 정리해 보고, 이들의 장단점을 파악하도록 한다. 또한, 대표적인 소프트웨어와 하드웨어 구조를 혼합하고, 병렬적으로 구성하여 성능을 높일 수 있는 새로운 혼합 구조를 제안한다. 성능 평가 결과는 제안된 구조가 lookup 속도를 향상시키면서도, 메모리 사용량도 줄일 수 있다는 것을 보여준다.

Abstract

When designing network processors or implementing network equipments such as routers are implemented, IP lookup operations cause the major impact on their performance. As the organization of the IP address becomes simpler, the speed of the IP lookup operations can go faster. However, since the efficient management of IP address is inevitable due to the increasing number of network users, the address organization should become more complex. Therefore, for both IPv4(IP version 4) and IPv6(IP version 6), it is the essential fact that IP lookup operations are difficult and tedious. Lots of researches for improving the performance of IP lookups have been presented, but the good solution has not been came out. Software approach alleviates the memory usage, but at the same time it is slow in terms of searching speed when performing an IP lookup. Hardware approach, on the other hand, is fast, however, it has disadvantages of producing hardware overheads and high memory usage. In this paper, conventional researches on IP lookups are shown and their advantages and disadvantages are explained. In addition, by mixing two representative structures, a new hybrid parallel architecture for fast IP lookups is proposed. The performance evaluation result shows that the proposed architecture provides better performance and lesser memory usage.

Keywords : High-speed IP lookup, Prefix, LPM(Longest Prefix Matching), Router, Low memory usage, Multiple CAM, Ternary CAM

* 正會員, 延世大學校 電氣・電子工學科

(Dept. of Electrical Eng., Yonsei Univ.)

接受日字:2003年3月5日, 수정완료일:2003年4月30日

Ⅰ. 서 론

인터넷 사용자의 폭발적인 증가로 인하여, 호스트수가 엄청나게 늘어나고 있고, 그에 따라서, 네트워크의 수도 매우 빠른 속도로 늘어나고 있다. 이러한 추세로 인하여 기존 IP 어드레스가 부족하게 되었고, 그 결과 새로운 IP 어드레스 체계인 IPv6가 개발 중이고, 기존의 클래스 개념이 무너져, 이른바 CIDR(Classless Inter-Domain Router) 어드레스 체계가 도입된지도 꽤 많은 시간이 흘렀다. 이러한 어드레스 체계의 확장은 IP 어드레스를 보고 해당 네트워크로 보내는 IP lookup 동작 수행을 복잡하게 만들었다. 그러나, 사용자들의 요구가 커지고, 멀티미디어적인 응용 분야가 많아짐으로 인해서, 많은 패킷들을 보다 빠르게 전송해야 할 필요성이 증가하고 있다. 패킷 전송망에서 라우터들의 역할이 트래픽이 적은 경로로 패킷들을 전송하는 것이고, 여기서 가장 시간이 많이 소요되고 어려운 작업 중 하나가 IP lookup 연산이다.

IP lookup 연산이 어려운 이유로는, 첫째로 네트워크 부분으로 정해진 길이가 없다는 것이다. IPv4의 경우 32-bit의 길이를 갖는 IP 어드레스를 사용한다. 이 32-bit의 길이는 다시 NI(Network Identifier)와 SI (Station Identifier), 두 부분으로 나누어진다. 전자는 보내려고 하는 station이 속해 있는 서브 네트워크를 찾아갈 수 있도록 하는데 사용되고, 후자는 전자에 의해 찾아진 서브 네트워크 내에서 정확한 station을 찾아가기 위한 MAC(Medium Access Control) 어드레스를 생성하기 위해서 사용된다. 예전의 클래스에 기반한 어드레스 체계에서는 각 클래스별로 정해진 NI 부분을 가지고 있어 lookup 동작이 용이하였으나, 어드레스의 비효율적인 사용을 이유로 점차 CIDR 방식을 사용하게 되었다. CIDR에서는 정해진 NI 길이가 없기 때문에 IP lookup 동작이 매우 어렵다. 둘째로, IP 어드레스가 계층성(Hierarchy)을 갖는 이유로 더욱더 IP lookup 동작이 어려워지고 있다. 예를 들면, NI가 165.X.X.X처럼 앞의 8-bit가 NI인 네트워크가 있을 수도 있고(이 경우에 165, 즉 이진수로 10100101을 prefix라고 부른다.), 그보다 더 작은 네트워크가 165.132.X.X처럼 앞의 16-bit를 NI로 갖는 네트워크가 있을 수 있다. 이 경우 IP lookup 동작은 앞의 165만을 찾았다고 해서 끝나는 것이 아니라, 165.132와 일치하는지도 조사하여야 한다. 여러 개의 NI와 일치하는 경우에는 가장 긴 prefix와 매치하는 것을 찾아야 하는데, 이 방식을 LPM(longest prefix matching) 방법이라고 한다[1]. 셋째로, NI 부분을 찾아내기 위한 prefix를 저장하기 위해 사용되는 메모리의 한계로 인해 IP lookup 동작이 어려워지고 있다. 메모리의 사이즈를 늘리면, 보다 많은 엔트리를 저장할 수 있는 것은 장점이지만, 메모리와 프로세서와의 속도차이나 메모리 내에서 원하는 prefix를 찾아내는데 걸리는 시간으로 인하여 성능은 떨어질 수 있다.

이 논문에서는 IP lookup 속도를 향상시키기 위한 기존의 방법들에 대해서 설명하고, 그 구조들을 병합하여 성능을 향상시킨 혼합적인 IP lookup을 위한 병렬 구조를 제안한다. 성능 평가 결과 제안된 구조가 기존의 방법보다 빠른 lookup 동작을 수행함을 검증하였다. 이 논문의 순서는 다음과 같다. Ⅱ장에서는 기존 연구들의 소개와 각각의 장단점에 대해서 언급하고, Ⅲ장에서 이 논문에서 제안하는 병렬 구조에 대해서 소개하며, Ⅳ장에서는 성능 평가 결과를 보이고, Ⅴ장에서 결론을 짓도록 한다.

Ⅱ. IP lookup 속도 향상을 위한 기존 연구들

기존의 연구는 하드웨어적인 접근방법과 알고리즘을 적용한 소프트웨어적인 방법으로 나누어 질수 있다. 하드웨어적인 접근방법은 빠른 검색 속도를 가지고 있지만 대신 하드웨어 오버헤드가 크다는 단점이 있고 반면 소프트웨어를 사용한 방법은 하드웨어적인 복잡성이 줄어들고 유연한 확장성을 제공해 주지만 검색 속도가 느리다는 단점이 있다. 이러한 하드웨어적인 접근방법과 소프트웨어 적인 방법의 장단점을 잘 파악하고 이의 조화가 앞으로의 주요한 연구 과제가 되리라고 생각된다. 먼저, 대표적인 하드웨어적인 접근 방법으로 Gupta[2]가 사용한 방법을 살펴보았다. Gupta의 연구에서는 하드웨어적인 방법을 사용하여 소프트웨어적인 방법보다 속도를 향상시키는데 주안점을 두었다. <그림 1>은 Gupta의 방법 중에서 가장 기본적인 개념을 나타내고 있다. 즉 32-bit의 IP 어드레스 중에서 상위 24-bit에서 prefix를 찾기 위해 하나의 테이블인 TBL24를 두고, 이 테이블에서 가장 길게 일치하는 prefix를 찾지 못할 경우를 대비해서 나머지 8-bit을 위한 테이블인 TBLlong을 하나 더 두게 된다. TBL24의 contents 부

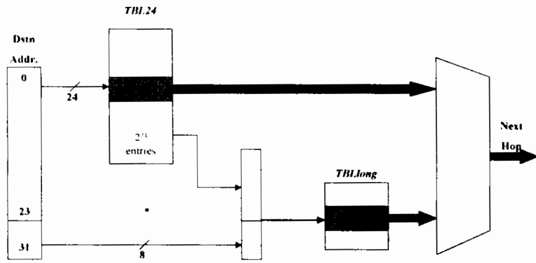

그림 1. DIR-24-8-BASIC 구조
Fig. 1. DIR-24-8-BASIC structure.

분에는 그 엔트리의 valid bit과 출력포트 정보가 있다. 만약 prefix가 상위 24-bit보다 긴 것에 해당하는 엔트리가 TBL24에 있으면, 그 엔트리의 출력포트 정보는 TBLlong이라는 두 번째 테이블의 어드레스를 가리키는 포인터가 된다.

Gupta의 방법은 하드웨어적인 구현으로 leaf pushing 방식으로 빠른 IP lookup 동작을 수행할 수는 있지만, 또한 leaf pushing 방식이 가지는 한계로 인하여 메모리가 비효율적으로 사용되고, 새로운 엔트리의 업데이트가 매우 어려운 단점이 있다. <그림 1>의 기본적인 구조를 변형하여 여러 가지 개선사항을 논하고 있지만, 결국은 <그림 1>이 가진 한계를 벗어나지는 못한다. IP 어드레스는 최대 32-bit의 깊이(depth)를 갖는 이진수이므로, IP lookup의 가장 기본적인 해결책은 이진 검색이라고 할 수 있다. 그러나, 이진 검색으로는 시간이 너무 오래 걸리기 때문에, 대부분의 소프트웨어적인 접근에서는 해시 함수를 잘 정의하여 시간을 줄이려는 노력을 하고 있다. McAuley[3]은 클래스 기반에 맞는 평면적인 어드레스에 맞는 CAM 구조와 CIDR에 맞는 계층적 어드레스에 맞는 CAM, 그리고, 기타 다른 lookup 동작에 맞는 CAM의 구조들을 보여주고 있다. CAM은 이상적으로 가장 빠른 IP lookup 동작을 제공할 수 있는 하드웨어라고 할 수 있지만, 가격이 너무 비싸다는 단점이 있다. CAM은 라우팅 테이블을 고속으로 검색하기에는 최적의 메모리 구조라 할 수 있으나, 가격이 비싸므로, 경제적인 면에서는 매력이 떨어진다. IP 어드레스의 이진 검색을 고속으로 수행하기 위해서, 이진 트리에서의 여러 가지 압축기법이 소개되었다[4]. Nilsson[5]은 트리의 사이즈를 줄이기 위해서 수준 압축(level-compression)이라는 방법을 사용하였다. 이 방법은 EXTRACT()라는 함수를 사용하여 IP 어드레스의 특정 부분을 추출하여 수준 압축된 트리 정보를 찾아나가는 방법이다. 이 방법은 효과적인 압축기법을 사용해서 트리의 정보를 줄이고, 효과적으로 검색하도록 해준다. 그러나, 역시 테이블 안에 압축이 안되고 계층성을 갖는 prefix들이 여러 개가 존재할 경우에는 성능이 떨어질 수 있다는 단점이 있다. 이에, Eatherton[6]은 다음의 3가지 아이디어를 기초로 하여 lookup 알고리즘을 고안하고 이에 기초한 하드웨어 구조를 제안하였다. 첫번째 아이디어는 트리 비트맵을 구현하는데 있어서 모든 자식 노드를 연속적으로 저장하고 두 번째로는 leaf pushing을 하지 않기 위해 자식 노드의 정보를 포함하는 확장 패스 비트맵과 더불어 트리내의 prefix 정보를 저장하는 내부 비트맵을 구성하는 것이다. 마지막으로는 하나의 트리의 사이즈를 메모리 액세스 횟수를 고려하여 적당하게 결정하는 것인데 트리의 사이즈는 내부 비트맵과 확장 비트맵, 그리고 자식 노드에 대한 포인터 정보의 합이다. 검색 순서는 먼저 확장 비트맵을 이용하여 다음 자식 노드를 결정하고 내부 비트맵을 통하여 현재까지의 BMP(Best Matching Prefix)를 저장한 후 이동하는 방식이다. Pao[7]가 제시한 방법은 트리를 구현하는데 있어서는 [6]의 방법과 매우 유사하나 이를 하드웨어 구조로 변환한 방식으로 전체 이진 트리를 <그림 2>와 같이 8-bit non-overlapping 영역으로 분리하여 8-bit 각 서브트리를 트리 벡터와 마스크 벡터, 그리고 라우팅 벡터로 분리하여 메모리에 저장하고 입력 목적지 어드레스에 따라 트리 벡터와 마스크 벡터를 AND 조작하고 이에 대한 결과 벡터를 이용하여 라우팅 벡터에서 출력 포트 정보를 취하는 방식이다.

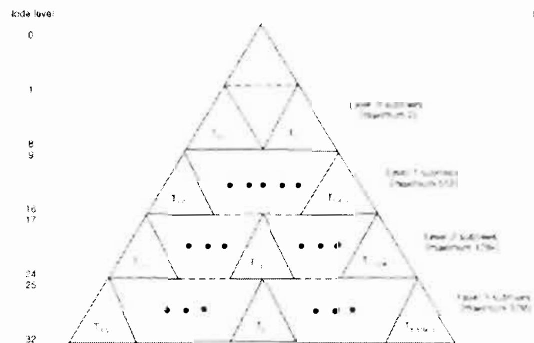

그림 2. 이진 트리를 non-overlapping 255 노드 서브트리로 분할
Fig. 2. Partitioning of the binary tree into non-overlapping 255-node subtrees.

8-bit씩 분리되어 총 4단계로 분리된 서브 트리들 가운데 상위 2단계까지는 상기 방식을 사용하고 3단계에서는 트리 벡터와 더불어 마지막 4단계를 위한 추가 포인터 정보를 삽입하여 4단계 서브트리 정보를 나타낸다. Pao의 구조는 메모리 사이즈를 최소한으로 희생하고 병렬처리와 파이프라이닝 하여 최대한의 빠른 lookup을 하려는데 초점을 두고 있다. 그러나, Pao가 제안한 방법은 ILP(Instruction Level Parallelism)구조가 아닌 일반적인 하드웨어적인 데이터 흐름을 5단계로 나누어 파이프라이닝하기는 쉽지 않으며 또한 병렬처리를 위한 하드웨어 오버헤드가 적지 않다는 단점이 있다. 지금까지 상술한 하드웨어적인 접근방법에 더하여 다양한 소프트웨어적인 접근방법이 제안되었고 그 한 예가 적당한 알고리즘을 사용하여 테이블 사이즈를 줄여서 캐시에 넣을 수 있도록 하는 방법이 제안되었다. 캐시에 테이블을 위치시키면, 200MHz의 Pentium Pro나 333MHz Alpha21164 프로세서는 초당 수백만 lookup 연산을 수행할 수 있다. Degermark[8]는 traffic 지역성(locality)이 없다고 가정하고 캐시 안에 들어갈 수 있는 작은 라우팅 테이블을 구성할 수 있도록 하는 방법을 제시하였다. 고성능의 lookup 연산을 목적으로 한 것이 아니라, small table을 사용해서 lookup을 수행하도록 하는 것이 목적이다. 그러나 캐시 안에 테이블을 위치시킬 수 있다면, 자연히 고성능이 따라온다는 대전제 하에서 연구를 진행하였다. 또한, 캐시 안에 테이블을 위치시키는 것은 별도의 하드웨어 없이도 lookup 연산을 수행할 수 있다는 것을 의미하기도 한다.

Degermark는 small table 사이즈를 유지하기 위해 코드 워드 배열과 베이스 인덱스 배열을 사용하고 있다. <그림 3>과 같이 코드 워드(code word)배열은 10-bit 값과 6-bit offset으로 구성되어 있고, 베이스 인덱스(base index)배열은 하나의 코드 워드 배열 중에 1의 개수를 세서 저장하는 역할을 한다. <그림 3>은 IP 어드레스가 들어왔을 때, small table에서 원하는 출력포트를 찾아내는 과정을 나타낸 것이다. 이 방법은 캐시 안에 테이블을 위치시킴으로써 성능을 극대화시킨다는 취지는 매우 바람직하다. 그러나 트리 기반의 구조의 취약성을 극복하기에는 미진하여 최악의 경우에는 역시 트리 구조와 동일한 성능을 보인다.

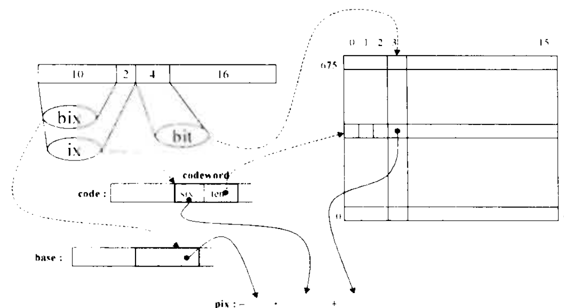

그림 3. 포인터 인덱스를 찾는 방법
Fig. 3. The method of finding the pointer indexes.

Ⅲ. 제안된 hybrid parallel 구조

기존에 제안된 방법으로는 크게 압축 알고리즘을 사용한 소프트웨어적인 방법과 성능을 중요시하는 하드웨어적인 접근방법이 있다. 각각의 장단점을 찾아내어 적절한 타협점을 찾아내기 위해서는 현재의 IP 어드레스에 대한 분석과 앞으로의 확장성에 대한 분석이 필요하다. 현재 IP 어드레스는 32-bit 이며 일반화하여 하드웨어 메모리 체계에 적용하면 2^{32}의 메모리가 필요하다. 실제 마이크로프로세서를 예로 들면 32-bit의 어드레스를 전부 물리 메모리에 매핑시킬수는 없으며 이에 따라 효율적으로 메모리를 관리하며 속도가 느린 주변장치로 인한 병목 형상을 처리하기 위하여 가상 메모리의 개념을 사용하고 이를 하드웨어적인 방법인 캐시 메모리와 OS가 관리하는, 즉 소프트웨어적인 방법인 MMU를 이용하는 기술을 발전시켜왔다. 이와 비슷한 개념으로 라우터나 네크워크 프로세서와 같이 IP lookup의 빠른 속도를 보장하기 위해서는 IP 어드레스의 현재 상황을 분석하여 IP 어드레스를 효율적으로 관리하고 활용하는 방안이 필요하다. 현상황하에서의 IP 어드레스 체계에서 prefix의 분포를 살펴보면, <그림 4>에서 IP 어드레스의 16에서 24-bit 사이에 대부분의 prefix 정보들이 분포되어 있다는 것과 24-bit보다 긴 prefix는 거의 존재하지 않는다는 것을 확인 할 수 있다[9]. IP 어드레스의 부족과 서브-네트워크 수의 증가로 인해서 24-bit보다 긴 prefix가 늘어날 전망이지만, IPv6의 등장과 이의 IPv4와의 호환성으로 인해 24-bit보다 긴 prefix는 큰 폭으로 늘어나지는 않을 전망이며 <그림 4>에서와 같이 25-bit, 26-bit, 27-bit의 prefix가 증가할 경우의 확장성에 대하여 확장 forwarding table에서 구현하였다.

이러한 prefix 분포를 바탕으로 고속 IP 어드레스

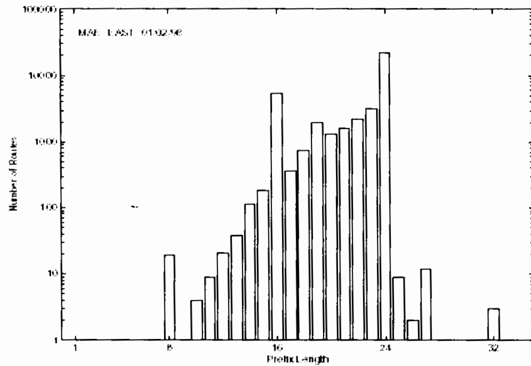

그림 4. prefix의 99%이상이 1-24bit 사이에 편중되어 있는 예

Fig. 4. An example that more than 99% of prefixes are from 1-bit to 24-bit.

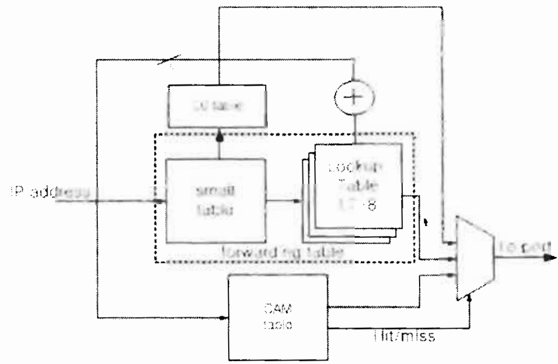

그림 5. Hybrid parallel 구조의 데이터 흐름

Fig. 5. The data flow of hybrid parallel architecture.

lookup을 수행하고 한정된 자원을 효율적으로 이용할 수 있는 구조를 고려해 본다. <그림 4>와 같이 25에서 32-bit 사이에 존재하는 prefix들은 나머지 부분에 존재하는 prefix들에 비해 상대적으로 적다. 물론, 전자와 후자간에 계층성을 가질 경우는 얼마든지 존재하므로 계층성을 갖는 경우를 무시할 수는 없다. 이러한 특징을 사용하면, IP 어드레스를 크게 두 부분으로 나눌 수 있다. 1에서 24-bit 사이의 길이를 갖는 prefix들과 그보다 긴 길이를 갖는 prefix들이다. 네트워크 프로세서를 설계하거나 라우터와 같은 장비의 시스템을 구축할 때, 마이크로프로세서는 반드시 들어가게 된다. 이렇게 내장되거나 시스템에 내장되는 마이크로프로세서의 역할은 소프트웨어적인 요소를 수행하고, 메모리와 같은 주변 블록들을 제어하는 것이다. 여기서 소프트웨어적인 요소에는 패킷을 분류하고, 네트워크 계층상의 내부 함수들을 처리하는 것 등을 모두 포함할 수 있다. 이 논문에서 제안하는 구조의 가장 중요한 설계 목적 중에서 병렬처리로 인한 IP lookup 동작의 향상과 메모리 사용량의 감소로 인한 캐시메모리의 활용을 최우선으로 하여 설계하였다. 설계시 메모리 사용량의 감소는 소프트웨어적인 방법인 small table을 사용하였고 또한 성능 향상을 위하여 Lookup Table LT-8 으로 하드웨어적인 방법을 채택하여 소프트웨어와 하드웨어의 장점을 취하는 구조를 고려하였다.

제안하는 구조는 <그림 5>와 같이 small table과 LT-8으로 구성된 forwarding table과 CAM table을 사용하여 병렬적으로 IP 어드레스를 처리하고 있는 과정을 보여주고 있다.

먼저 forwarding table에서는 IP 어드레스 1에서 24-bit를 처리하며 이와 병렬적으로 CAM table에서는 미리 정해진 entry 사이즈가 25-bit 이상인 IP 어드레스를 처리하여 만약 히트이면 CAM Table에서 정해진 출력 포트를 따르고 만약 히트가 아니면 forwarding table에서 처리된 출력 포트를 따르게 된다. forwarding table을 자세히 살펴보면 <그림 6>과 같은 구조를 가지고 있다. <그림 6>은 소프트웨어 접근 방법인 Degermark의 small forwarding 방법을 개량하여 small forwarding이 가지는 문제점인 16이상의 엔트리 비트를 가지고 있을때 메모리 참조가 많아지고 복잡해지는 부분을 forwarding table 에서는 LT-8으로 해결하였다. 먼저 16-bit까지의 prefix entry는 small table을, 17-24까지의 prefix entry는 LT-8을 사용한다. LT-8의 256개의 entry는 모두 확장된 leaf 노드이며 이것은 다음 포인터를 가지지 않고 순수한 출력 포트정보만 가지고 있다. 이렇게 함으로써 Gupta가 가지고 있는 광범위한 업데이트 문제를 작은 사이즈의 LT-8으로 한정할 수 있으며 또한 Degermark의 문제점의 하나인 복잡한 소프트웨어적인 검색 연산과 parse, dense, very dense로 나누는 모호함과 이로인해 소요되는 최적화 작업을 줄일 수 있다. 또한, 완전한 prefix을 사용함으로 인해 LT-8의 개수를 줄일수 있으므로 결론적으로 메모리를 효율적으로 사용할 수 있게 된다.

그러나 forwarding table을 사용하면 현재 prefix entry 상황 하에서는 잘 적용 되지만 앞으로 증가 추세에 있는 25-bit 이상의 prefix를 처리하는 데에 있어서 또 다른 확장성이 요구 된다. 증가하고 있는 24-bit를 넘는 prefix를 처리하기 위해 CAM 메모리를 더 많이

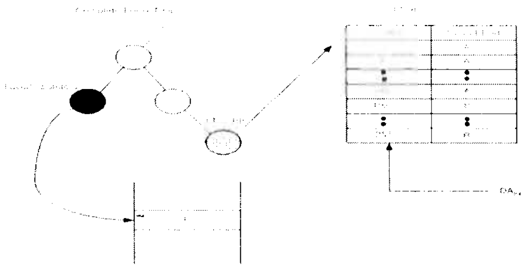

그림 6. LT-8 Table을 포인팅하고있는 complete prefix tree
Fig. 6. Complete prefix tree pointing LT-8 Table.

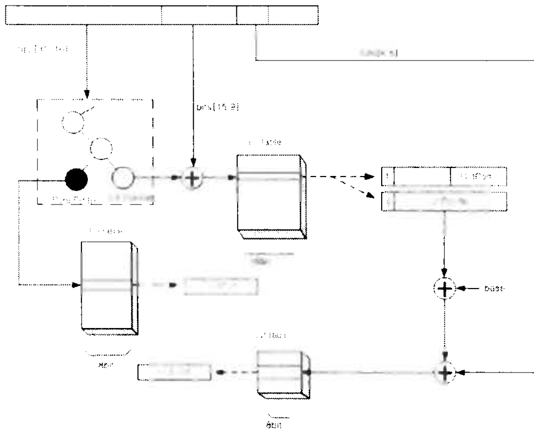

그림 7. 확장 forwarding table의 데이터 흐름도
Fig. 7. The data flow of expanding forwarding table

사용하는 것은 낭비적인 부분이 있기 때문이다. 이를 해결하기 위해 LT-8을 확장하여 <그림 7>과 같은 L1 table과 L2 table을 이용하는 방법을 제시한다.

제시하는 방법은 prefix entry의 수가 <그림 4>의 경우에서 25-bit, 26-bit, 27-bit 로 증가할 경우 L1 table과 L2 table의 2단계로 나누었다. 메모리 참조의 수를 줄이기 위해서는 L1 table의 사이즈를 8로 L2 table의 사이즈를 3으로 나누는 것이 좋으나 메모리의 효율적인 사용을 함께 고려한다면 7과 4로 나누는 것이 합리적이다. 따라서 L1 table과 L2 table을 각각 7-bit와 4-bit를 오프셋으로 사용하였다. 확장 forwarding table의 데이터 흐름을 살펴보면 처음 IP 어드레스의 상위 16-bit가 small table에서 순수한 prefix와 L1 포인터로 나뉘어 진다. 순수한 prefix라면 포트 정보가 출력포트로 나가게 되고 L1 포인터라면 7-bit 오프셋과 더해져서 L1 table의 계산된 위치에서 포트 정보를 찾게 된다. L1 table에서는 MSB 비트로 순수한 prefix인지 포인터인지를 구별해준다. 이와 같이 확장 forwarding table을 사용시에는 한번의 부수적인 메모리 참조를 하므로 속도의 저하가 있을 수 있지만 L1과 L2 table을 파이프라이닝한다면 상기 forwarding table의 LT-8과 같이 한 번의 메모리 참조로 구현 가능하다. 이를 종합하여, 확장 forwarding table은 <그림 8>과 같은 알고리즘으로 요약될 수 있다. <그림 8>의 알고리즘을 개략적으로 설명하면 확장 forwarding table 알고리즘은 small table에서 계산된 POINTER가 L1_Pointer인지 아니면 Pure_Prefix인지 확인한다. 만약 Pure_Prefix라면 L0_Table에서 원하는 출력포트로 IP 어드레스가 나가게된다. 만약 L1_Pointer라면 MSB가 1인지 0인지 확인하고 1이면 7bit 오프셋과 더해져서 L1_Table을 검색하여 원하는 출력포트로 나가게 되고 MSB가 0이면 base_addr와 4bit 오프셋과 더해져서 L2_Table을 검색하여 원하는 출력포트로 나가게 된다.

<그림 6>의 LT-8을 사용한 forwarding table이나 <그림 7>의 L1 과 L2 table을 사용한 확장 forwarding table은 정해진 prefix까지 IP 어드레스를 처리해준다.

```
IX <- DA31-20                    //IX: IP 어드레스의 상위 12bits
BIX <- DA31-22                   //BIX: IP 어드레스의 상위 10bits
BIT <- DA19-16                   //BIT: IP 상위 16bits의 하위 4bits
Codeword <- code[IX]             //SIX: Codeword에서 상위 6bits
TEN <- Codeword9-0               //DA: 목적지 어드레스
SIX <- Codeword15-10             //MSB: 최상위 비트
 PIX <- base[BIX] + six + maptable[TEN][BIT]

 POINTER <- level_pointers[pix]

 if(POINTER == Pure_Prefix)
 {
  OUTPORT <- L0_Table[POINTER]
 }
 else if(POINTER == L1_Pointer)
 {
  if(MSB == 1)
  {
  OUTPORT <- L1_Table[POINTER][DA15-9]
  }
  else
  {
  L2_Pointer <- base_addr + L2_Pointer
  OUTPORT <- L2_Table[L2_Pointer][DA8-6]
  }
 }
 end
```

그림 8. L1 table과 L2 table을 사용한 확장 forwarding table 알고리즘
Fig. 8. Expanded forwarding table algorithm using L1 table and L2 table

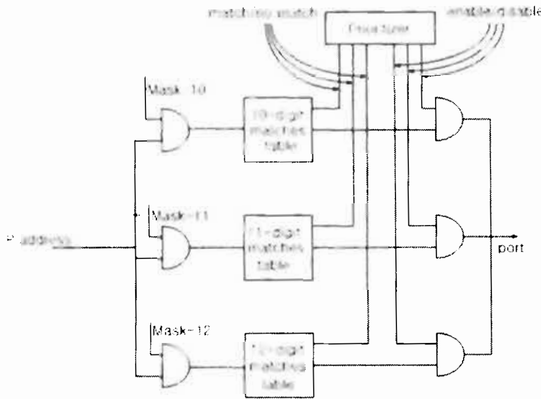

그림 9. CAM Table을 사용한 lookup operation
Fig. 9. Lookup operation with CAM Table.

그 이상으로 match되는 prefix entry는 CAM table을 사용하여 적은 사이즈의 CAM과 메모리의 사용으로 전체적인 IP lookup 속도를 향상시키려고 고안하였다. <그림 9>는 CAM table의 내부 구조를 나타내고 있다. <그림 9>에서 어드레스 계층에서 깊이에 따른 다른 CAM을 사용한다. 즉 디지트(digit) 어드레스가 10일때는 모든 10의 디지트를 저장한다. 따라서 IP 어드레스가 264.132.131.124라는 어드레스가 들어 갈 때 mask-10은 FFF-FFF-FFF-F 로 셋이 된다. 따라서 264.132.131.1xx의 어드레스와 그와 해당하는 출력 포트가 결정되게 된다. 이와 함께 Prioritizer가 선별하여 출력 포트로 보내게 되며 hit/miss 신호를 동시에 내보내게 된다.

제안하는 구조는 검색을 하는 부분에 있어서 메모리 사이즈 측면에서 부담이되는 25-bit이상의 prefix는 CAM table을 사용하고 24-bit 이하의 prefix는 적은 사이즈로 유지하여 이부분을 1차 캐시와 2차 캐시에 모두 넣을 수 있도록하여 하드웨어적인 부담을 줄일 수 있다. 더 나아가 마이크로프로세서의 공정과 구조의 변화로 인해 빨라지는 클럭 속도에도 유연하게 적용될 수 있도록 하는 데에도 그 목적이 있다.

Ⅳ. 구현 및 성능 평가

메모리 사이즈측면에서 살펴보면 small table단계에서는 codeword array의 8k, 그리고 base array indices의 2k, 그리고 maptable의 5.3k가 필요하다. 따라서 small table단계에서는 총 15.3k의 메모리 사이즈가 필요하다. 시뮬레이션 환경에서는 24-bit 이하의 prefix가

표 1. 메모리 사이즈 비교
Table 1. comparison of memory size.

Router	total no. of prefixes	no of prefixes with ≥ 25bits	Gupta[2]	Degermark[8]*	The proposed architecture
			memory(KB)	memory(KB)	memory(KB)
aads	29831	102	7000	753	128.1
mae-east	23729	103	7000	115	104.9
mea-west	34230	94	7000	132	144.9
pacbell	41811	139	7000	226	174
paix	16445	155	7000	None	96.7

99%라는 것을 고려하여 LT-8 을 사용하였다. LT-8단계에서 Lookup Table의 사이즈는 (# of LT-8s) × 2^8가 필요하다. 따라서 entry의 사이즈가 <표 1>의 pacbell의 경우41811으로 12.5k + 15.3k + (# of LT-8s) × 2^8의 총 메모리가 필요하다. (여기서 12.5k는 40k의 30%를 가정하였다.) # of LT-8s의 대략적인 사이즈를 계산하기위해 41811 entry중 16-bit 이하30%와 25-bit 이상의 prefix를 빼면 29248의 17~24bit의 prefix entry가 계산되고 이중 20%정도가 LT-8을 채운다고 가정하면, (29248/(256*0.2)) * 2^8 으로 146.2k 의 메모리가 필요하다. 따라서 총 (146.2k + 15.3k + 12.5k) 로 174k의 메모리 사이즈만 필요하게 된다. <표 1>을 살펴보면 mea-west의 경우 제안한 구조보다 Degermark의 메모리 사이즈가 더 적게 나타나는데 이것은 mea-west처럼 leaf node들과 next hop의 수가 적은 경우에는 제안하는 구조보다 적은 메모리 사이즈를 보여줄 수 있다.

또한 lookup 속도를 비교하기 위해서 기존 논문의 방법을 참조하여 Pentium Pro의 시스템으로 환산하여 시뮬레이션하였다. 1차 캐시의 사이즈는 8KB이고 latency가 5ns, 2차 캐시의 사이즈가 256KB이고 latency가 15ns, on chip SRAM의 사이즈가 2MB이고 latency가 35ns, 그리고 DRAM access time이 60ns인 시스템에서 Degermark의 방법은 최악의 경우 101cycle로 400MHz인 경우 202ns의 시간이 걸린다. 또한 모든 데이터가 1차와 2차 캐시에 있다고 보면 69cycle로 172.5ns인 평균적인 lookup time이 소비된다. Gupta의 방법은 선형적으로 메모리에 의존하고 있으므로 DRAM을 사용하게 된다. 따라서 하드웨어의 복잡성을 증가시키고 메모리를 파이프라이닝 한다면 한번의 메모리 access time (60ns)만 걸릴 것이다. 그러나 하드웨어의 복잡성을 피하기 위해 파이프라이닝 하지 않는다면 3번의 메모리

표 2. 제안된 구조와 기존의 구조와의 lookup time 비교
Table 2. Comparison of the proposed architecture with existing schime

	Gupta[2]	Degermark[8]	The Proposed Architecture
Lookup Time(ns) for 24 Prefix	60ns (Pipeline)	172.5ns	77.5ns
Worst Case(ns) Lookup Time	180ns (non Pipeline)	202ns	115ns

access time(180ns)이 걸릴 것이다. 제안하는 구조는 최악의 경우 8*4+14로 46cycle, 즉 115ns의 시간이 걸리며 평균적으로 L1의 miss rate이 4%, 그리고 인스트럭션당 1.5의 메모리 참조가 있다고 가정하면 8*(2.5+4%*(15+50%*24))로 29cycle, 즉 72.5ns가 소비된다. 실제로 시뮬레이션한 결과는 예측치와 비슷한 31cycle이 나왔다. 만약 확장 forwarding table을 사용한다면 한번의 메모리 액세스를 더 하게 되지만 파이프라이닝을 한다면 속도면에서는 forwarding table과 같게 나올 것으로 생각된다. <표 2>는 시뮬레이션한 결과를 비교하였다.

앞서 언급한대로, Degermark의 방법은 메모리 사용량을 줄이고, 캐시를 사용하는 구조를 택함으로써 성능향상을 꾀하였지만 언제나 캐시에 히트가 된다는 점을 보장하지는 못한다는것과 하위 메모리를 검색할 때 빈번한 메모리 참조가 발생할 수 있다는 단점이 있고, Gupta의 방법은 하드웨어를 사용하여 속도를 높였으나, 메모리 사용량이 너무 큰 단점이 있다. 즉, Gupta의 방법은 leaf pushing 방법으로 인하여 IP lookup시 검색을 빨리 할 수 있는 장점이 있지만 이로인한 메모리 사용량이 커지게 된다. CAM을 사용한 라우팅 lookup은 가장 속도가 빠르지만 CAM의 본질적인 문제인 고비용이라는 단점이 있다. 제안한 구조는 메모리 사용량도 줄이고 성능도 향상시키면서, 가격면에서도 그렇게 큰 증가를 보이지 않게 하기 위함이 목적이었다. <표 1>과 <표 2>에서 알 수 있듯이, 제안된 구조가 Degermark의 방법보다 메모리 사용량이 비슷하거나 줄이면서도, 성능을 높일 수 있다. Gupta의 방법을 파이프라이닝하였을 때보다는 성능이 떨어지기는 하지만, 파이프라이닝으로 인한 하드웨어적인 오버헤드를 피하고 메모리 사용량을 월등히 줄임과 동시에 캐시 메모리를 활용하여 보다 빠른 lookup을 가능하도록 하였다. 제안된 구조는 24-bit 이하의 테이블만을 구성하므로, 캐시 안에 넣을 테이블의 사이즈는 상대적으로 줄어들게 된다. 이로 인해, 캐시 안에 넣을 수 있는 가능성이 더 높아지게 되므로, 미스 패널티가 발생할 확률이 더욱 줄어들게 된다.

V. 결 론

제안된 고속 IP lookup 연산을 혼합 병렬 구조는 메모리의 사용을 최소화하면서도 비교적 높은 성능의 IP lookup 연산을 수행할 수 있다. 고속 IP lookup 연산은 네트워크 프로세서나 라우터와 같은 장비를 구현할 때, 가장 시간적인 고려를 많이 해야 하는 부분이면서도 하나의 해결책이 제시되지 못한다. 메모리 사용량이 많으면, 페이지 폴트(page fault)나 SoC로의 구현 불가 등의 여러 가지 문제가 발생할 수 있다. 당연히 하드웨어적으로 구현하는 것이 소프트웨어적으로 구현하는 것보다 속도가 빠르다. 결국은 적은 메모리를 사용해서 효과적인 하드웨어 구조를 가지고, 메모리를 접근하느냐하는 것이 고속 IP lookup 동작을 위한 필요조건이라고 할 수 있다. 고속 IP lookup 연산을 위해서 새로운 방법을 제안하는 것이 원론적으로는 가장 좋은 방법이라고 할 수 있겠다. 그러나, 모든 상황에서와 마찬가지로, 기존의 방법을 잘 조합하여 가격 대 성능비의 트레이드-오프(trade-off) 관계를 조절하여 원하는 해결책으로 만드는 것 또한 중요한 연구의 한 줄기라고 할 수 있다. 따라서 본 논문에서 제안한 구조는 IP lookup시 CAM의 사용을 최소화하고 forwarding table과 병렬적으로 처리하여 메모리 사이즈를 적게 하고 속도를 IP 어드레스의 상위 24비트로 한정하여 IP lookup 속도를 향상시켰다. 제안된 구조는 하드웨어적인 지원을 바탕으로 소프트웨어적으로 제어하면서 구현을 하였고, 병렬적인 구조를 채택함으로써 성능을 높일 수 있었다.

참 고 문 헌

[1] Rich Seifert, The Switch Book : The Complete Guide to LAN Technology, John Wiley & Sons, Inc., 2000.

[2] Pankaj Gupta, Steven Lin, and Nick Mckeown. "Routing Lookups in Hardware at Memory Access Speeds," Proceeding of Infocom 98, 1998, VOL.40, pp. 1240~1247

[3] A. McAuley and P. Francis. "Fast Routing Table Lookup Using CAMs," Proceeding of Infocom 93, 1993, VOL.3, pp. 1382~1391
[4] Ruiz-Sanchez, M.A.; Biersack, E.W.; Dabbous, W., "Survey and taxonomy of IP address lookup algorithms," IEEE Network, VOL.15, Mar/Apr 2001, pp. 8~23
[5] S. Nilsson and G. Karlsson, "Fast address lookup for Internet routers," Proceeding of IFIP 4th Int. Conf. Broadband Communications, Apr. 1998, pp. 11~22
[6] W. Eatherton, "Hardware-Based Internet Protocol Prefix Lookups," Master's thesis, Washington Univ., 1999
[7] D. Pao et al., "Efficient Hardware Architecture for Fast IP Address Lookup," Proceeding of IEEE INFOCOM '02, 2002, pp. 555~561
[8] DegerMark, et al. "Small Forwarding Tables for Fast Routing Lookups," Proceeding of ACM Sigcomm 97, 1997, pp. 3~14
[9] Internet Performance Measurement and Analysis Project, University of Michigan and Merit Network, URL : http://www.merit.edu/ipma

저 자 소 개

徐 大 植(正會員)
2000년 : 연세대학교 전기공학과 졸업. 2001년~현재 : 연세대학교 전기전자공학과 석사과정

吳 在 錫(正會員)
2001년 5월 : University of Bridgeport, Computer Engineering Major 졸업. 2002년~현재 : 연세대학교 전기전자공학과 석사과정

尹 晟 澈(正會員)
2000년 : 연세대학교 전기공학과 졸업. 2002년 : 연세대학교 전기전자공학과 대학원 졸업. 현재 : Mtekvision 주임 연구원. <주관심분야 : CAD 및 VLSI>

姜 成 昊(正會員)
1986년 2월 : 서울대 공대 제어계측공학과 졸업(학사). 1988년 5월 : The University of Texas at Austin 전기 및 컴퓨터공학과 졸업(공학석사). 1992년 5월 : The University of Texas at Austin 전기 및 컴퓨터공학과 졸업(공학박사). 1989년~1992년 : Schlumberger Inc. Research Scientist. 1992년~1994년 : Motorola Inc. Research Scientist. 1994년~1999년 : 연세대학교 공과대학 기계전자공학부 조교수. 1999년~현재 : 연세대학교 공과대학 기계전자공학부 부교수. <주관심분야 : 테스팅 및 DFT, CAD, SOC 설계>