논문 2006-43TC-5-17

적응적인 복수 해싱과 프리픽스그룹화를 이용한 고속 IP 주소 검색 구조

(A High-speed IP Address Lookup Architecture using Adaptive Multiple Hashing and Prefix Grouping)

박 현 태*, 문 병 인**, 강 성 호***

(Hyuntae Park, Byung In Moon, and ho Kang)

요 약

IP 주소 검색 구조는 라우터 시스템에서 고속 네트워크 기술의 중요한 이슈가 되고 있으며 패킷 전달의 성능을 좌우하는 주요한 문제 요소로 지적되고 있다. 본 논문에서는 복수 해싱의 적응적인 적용과 프리픽스 그룹화를 이용하여 효율적인 고속 IP 주소 검색 구조를 제안한다. 여러 라우팅 데이터의 엔트리 분포를 분석하여 프리픽스를 그룹화하고 그룹별로 적용되는 해쉬함수의 개수를 적응적으로 적용하여 해싱에 의한 충돌(collision)을 줄일 수 있었으며 이를 통해 테이블의 수를 최적화하고 메모리 효율을 높일 수 있었다. 또한 제안하는 구조는 단 한 번의 메모리 접근만으로 포워딩 테이블의 구성 및 검색 과정을 수행할 수 있는 고속 구조이다.

Abstract

IP address lookup has become a major bottleneck of packet forwarding and a critical issue for high-speed networking techniques in routers. In this paper, we propose an efficient high-speed IP address lookup scheme using adaptive multiple hashing and prefix grouping. According to our analysis results based on routing data distributions, we grouped prefix lengths and selected the number of hash functions in each group adaptively. As a result, we can reduce collisions caused by hashing. Accordingly, a forwarding table of our scheme has good memory efficiency, and thus is organized with the proper number of memory modules. Also, the proposed scheme has the fast building and searching mechanisms to develop the forwarding table only during a single memory access.

Keywords: IP Address Lookup, Adaptive Multiple Hashing, Prefix Grouping

I. 서 론

최근 인터넷을 비롯한 네트워크 기반의 데이터통신은 네트워크 트래픽의 엄청난 증가와 급변하는 환경에 대한 효율적인 대처가 필요하게 되었다. 이러한 상황에서 전체적인 네트워크 성능 향상을 위해서는 링크매체의 높은 대역폭(bandwidth), 즉 링크 전송속도(link speeds)와 라우터의 데이터효율(throughput), 패킷 전달율(packet forwarding rate)이 모두 우수해야 한다. 최근 링크매체의 전송속도와 대역폭은 광섬유 케이블의 출현으로 인해 크게 향상되었다. 또한 라우터의 데이터 효율은 멀티 레이어 스위칭 기술과 스위치 패브리케이션(switch fabrication)기술의 발전으로 크게 향상되었다. 이에 반해 상대적으로 패킷 전달율은 낮은 성능을 보이고 있으며 라우터의 주요 성능저하 요인이 되고 있다. 따라서 인터넷 라우터의 패킷 전달율을 향상시키기 위한 IP 주소 검색 기술이 중요시 되고 있다.[1]

* 학생회원, *** 평생회원 연세대학교 전기전자공학과
(Department of Electrical and Electronic Engineering, Yonsei University)
** 정회원 경북대학교 전자전기컴퓨터학부
(School of Electrical Engineering & Computer Science, Kyungpook National University)
※ 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음.
접수일자: 2005년12월22일, 수정완료일: 2006년5월15일

IPv4 주소체계에서 과거에는 32bit의 주소를 4개의 클래스로 나누어 각 클래스에 따라 고정된 프리픽스 길이를 결정하는 클래스 기반 방식을 사용하였다. 그러나 이 주소 체계는 클래스에 따라 계층구조로 이루어져 있기 때문에 IP 주소의 낭비가 심하였다.[2] 이를 보완하기 위해 CIDR(Classless Inter-Domain Routing) 이라는 주소 체계가 도입되었다.[3] CIDR에서는 클래스와 무관하게 8~32bit의 임의의 길이를 프리픽스로 사용한다. 따라서 프리픽스의 길이는 패킷마다 가변적이며 더욱이 패킷 자체에는 프리픽스의 길이에 대한 정보가 없기 때문에 주소 검색 동작에서 단순 exact 매치 방식으로는 검색을 할 수 없다. 따라서 가능한 모든 프리픽스 길이 중에서 가장 길게 매치 하는 프리픽스(longest prefix matching)를 검색이 필요하게 되었다.

LPM을 위한 새로운 IP 주소 검색 구조 중에서 빠른 검색속도를 가진 해쉉을 이용한 방식이 각광을 받고 있다. 해쉉을 이용한 검색에서는 서로 다른 키(key)값에 의한 해쉉의 결과가 일치하게 되는 충돌(collision)이 가장 문제 시 된다. 특히 복수 해쉉기법은 충돌 문제를 해결할 수 있는 좋은 해쉉 성능(good hashing)을 가지지만 적용하는 해쉬 함수의 수에 따라 메모리 크기가 배로 커지는 단점이 있었다. 또한 LPM검색을 위해 다른 프리픽스 길이마다 포워딩 테이블을 구성해야 하므로 별도의 테이블 수가 많아지고 이를 구현하기 위한 메모리 모듈의 수가 많아지는 문제점이 발생하였다.

본 논문에서는 프리픽스 그룹화 기법을 개선하여 적용하고 그룹별로 복수 해쉉을 적응적으로 적용하여 충돌의 발생 가능성을 줄이면서도 메모리 효율이 뛰어나도록 테이블을 구성한다. 또한 단 한 번의 메모리 접근만으로 포워딩 테이블의 구성 및 검색이 가능한 고속 IP 주소검색 구조를 제안하였다.

Ⅱ. 기존 해쉉 기반 IP 주소 검색 구조

초기 연구에서는 기존의 소프트웨어 기반 방식에 해쉉을 접목하여 메모리 크기를 줄이고 확장성을 보완하는데 주로 사용되었으나 최근에는 해쉉 자체를 바로 검색과정에 적용하여 포워딩 테이블을 구성하고 검색함으로써 더욱 메모리 접근을 줄인 고속 구조가 제안되고 있다. 대표적으로 프리픽스 길이에 따라 포워딩 테이블을 따로 구성하고 해쉉을 통해 구성 및 검색을 수행하며 충돌 발생 시에는 보조 테이블에 저장하여 바이너리 검색을 하는 구조가 제안되었다.[4] 그러나 이 방식은 지나치게 많은 테이블이 필요하며 충돌 발생 시에는 바이너리 검색을 해야 하므로 경우에 따라 긴 검색시간을 필요로 한다.

충돌을 충분히 피할 수 있는 완전 해쉬함수를 찾는 데는 긴 시간이 필요하므로,[5] 이 대신에 복수의 해쉬 함수를 이용하는 방법이 제안되었다.[6] 그러나 이 구조에서는 LPM을 위한 구체적 구현 방법이 제안되어 있지 않고 충돌 발생 확률을 감소시키는 효과는 얻었지만 낮은 발생 확률이지만 발생 시 이에 대한 대처 방안이 제시되어 있지 않다.

이를 개선하기 위해 프리픽스들을 길이별로 분류하여 각 길이마다 각각 복수의 포워딩 테이블을 구성하고 복수 해쉉을 통해 병렬 검색이 가능한 구조가 제안되었다.[7] 이 방식은 CRC-32 해쉉 하드웨어를 이용하여 각 길이별로 복수의 해쉬 인덱스 값을 구하여 각 테이블을 참조하였다. 참조된 각 버켓에는 두 개 이상의 로드가 존재하여 충돌 발생 시에도 여러 엔트리를 저장할 수 있도록 하였다. 또한 해당 버켓이 모두 오버플로우될 경우에는 작은 크기의 TCAM에 저장하였다. 따라서 모든 길이 별 테이블과 오버플로우 테이블을 동시에 병렬 검색하여 한 번의 메모리 접근으로 주소 검색을 수행할 수 있었다. 하지만 이 방법에는 크게 다음 4가지 단점이 있다.

첫째, 테이블 수가 크게 증가하여 개별적인 메모리 모듈이 많이 필요하게 된다. 모든 프리픽스 길이와 해쉬 개수에 따라 개별적인 포워딩 테이블이 구성되어야 하며 이는 개별적인 메모리 모듈로 구현되어야 한다. 따라서 테이블 수의 증가는 그 만큼 별도의 메모리 모듈이 많이 필요하다는 것을 의미하므로 구현이 복잡해진다. 이를 개선한 연구로서 프리픽스 그룹화 기법을 이용한 방식이 제안되었다.[8] 각 프리픽스 길이 마다 테이블을 할당하는 대신 여러 프리픽스를 적당한 길이로 묶어서 하나의 테이블에 저장함으로써 테이블 수를 줄이는 것이다.

둘째, 포워딩 테이블 구성 과정에서는 두 번의 메모리 접근이 필요하게 된다. 구성 과정은 복수의 해쉬 인덱스가 참조하는 여러 버켓의 모든 데이터를 읽어 들인 후 로드 수를 비교하여 더 여유 공간이 많은 쪽에 삽입하게 된다. 저장할 테이블의 버켓이 결정되면 해당 버켓의 기존 데이터와 저장할 데이터를 합쳐서 전체 버켓의 내용을 재구성한 후 해당 테이블의 버켓으로 다시 쓰기 작업한다. 이처럼 읽기와 쓰기 동작이 각각 필요하므로 두 번의 메모리 접근이 필요하게 된다.

셋째, 구성 동작 중 오버플로우 발생시에도 불필요한 위의 과정이 반복되어야 한다. 각 버켓에 몇 개의 엔트리가 있는지 정보는 각 버켓의 'Number of Items' 필드에 저장되어 있다. 따라서 오버플로우 상태를 알기 위해서도 일단 해당 버켓의 데이터를 먼저 읽어 들이는 작업이 필요하게 되므로 오버플로우 테이블을 구성하기 위해서도 두 번의 메모리 접근이 필요하게 된다.

마지막으로, 프리픽스 그룹화 기법을 적용할 경우, 각 그룹별 엔트리의 수와 분포가 크게 차이가 있기 때문에 모든 그룹에 대해 동일한 개수의 해쉬함수를 적용하는 것은 비효율적이다.

Ⅲ. 제안하는 IP 주소 검색 구조

1. 라우팅 데이터 분석 및 프리픽스 그룹화

그림 1은 백본 라우터의 프리픽스 길이별 엔트리 수의 분포이다. 라우팅 데이터는 각 라우터의 환경 및 시간에 따라 다를 수 있으므로 본 연구에서는 총 8가지의 다른 데이터를 고려하여 프리픽스 분석 및 실험을 하였다.

프리픽스 그룹화(prefix grouping)는 여러 프리픽스를 적당한 길이로 묶어서 하나의 테이블에 저장하고 해쉬을 적용하는 것을 말한다. 이 방식의 가장 큰 장점은 테이블 수를 줄임으로서 필요한 메모리 모듈의 수를 줄일 수 있다는 점이다. 또한 그룹별로 다른 테이블에 프리픽스가 분산되기 때문에 그 만큼 충돌의 발생 가능성도 낮아지는 효과가 있다. 물론 프리픽스 길이 별로 테이블 수가 존재한다면 늘어난 테이블 수만큼 충돌이 분산되는 효과도 더 커진다. 즉 적용하는 해쉬함수의 개수는 곧 필요한 테이블의 수와 비례하며, 테이블의 수

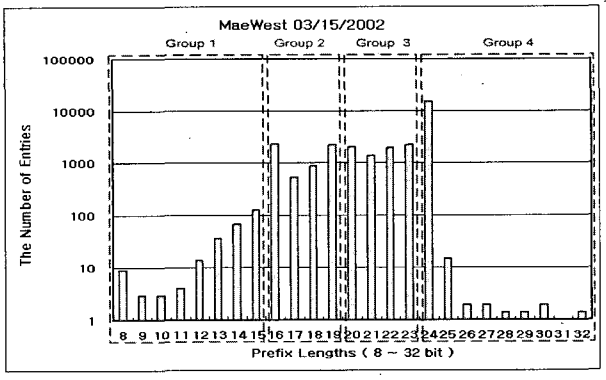


그림 1. 프리픽스분포 및 제안하는 프리픽스 그룹화
Fig. 1. The distribution of prefix and the proposed prefix groups.

와 충돌의 발생 확률은 trade-off관계를 가진다. 따라서 프리픽스의 통계적 분포에 따라 효과적으로 그룹화하면 적절한 테이블 수와 충돌 발생 확률 관계의 포화점(saturation point)에 근접할 수 있다.

본 연구에서 적용한 그룹화는 그림 1에서 함께 표시하였다. 그룹의 선택은 프리픽스 분포를 분석하여 다음 두 가지 사항을 고려하여 수행하였다. 첫째, 임의의 프리픽스 길이를 갖는 것끼리 묶는 것 보다 연속된 프리픽스 길이끼리 묶는 것이 용이하다. 그 이유는 연속된 bits입력에 의해서 인덱스 값을 생성하는 CRC-32를 해쉬함수로 사용하기 때문이다. 또한 분포를 고려할 때 임의의 프리픽스 길이끼리 특별한 연관성이 없으며 오히려 인접한 프리픽스 길이끼리 비슷한 엔트리 수를 갖는다는 점을 고려하였다. 둘째, 특정 그룹에서 가장 많은 엔트리를 갖는 프리픽스 길이가 각 그룹의 첫 번째 프리픽스 길이가 되도록 한다. 그룹별로 해쉬함수의 입력은 첫 번째 프리픽스 길이를 사용하기 때문에 프리픽스 분포가 많은 길이를 그룹의 시작 프리픽스로 함으로써 더 신뢰도 높은 해쉬 인덱스 값을 구할 수 있다. 이를 토대로 본 연구에서는 프리픽스를 8~15, 16~19, 20~23, 24~32인 4개의 그룹으로 나누었다.

2. 복수 해쉉의 적응적 적용

가. CRC-32를 이용한 해쉉

제안하는 구조에서 해쉬함수는 CRC-32 polynomial을 이용하여 구현하였다. CRC-32 polynomial은 충돌을 최소화할 수 있는 완전 해쉬함수(semi-perfect hash function)에 준하는 우수한 성능을 가지고 있으며 간단히 하드웨어로 구현이 용이하다. 또한 다양한 프리픽스 길이가 입력으로 쓰이더라도 이에 크게 영향을 받지 않고 일정하고 매우 높은 엔트로피를 갖으며 복수의 해쉬 인덱스를 추출하기에도 용이하다.

나. 복수 해쉉의 그룹별 적응적 적용

근본적으로 충돌은 해쉉을 통해 큰 데이터베이스를 더 작은 데이터베이스로 줄이거나 분산시킴으로서 발생하며 이 과정에서 많은 키(key)값을 훨씬 작은 인덱스 값으로 바꾸기 때문에 일어난다. 즉 key값이 많을수록 인덱스 값이 작을수록 충돌은 더 많이 발생하는 것이다. 인덱스 값의 범위는 할당하는 메모리 크기에 의해 결정된다. 따라서 일정한 메모리 효율에 대하여 충돌의 발생 확률은 해당 엔트리 개수에 비례한다고 볼 수 있다. 여기서 메모리 효율이란 각 테이블에 예상되는 엔

표 1. 그룹별 엔트리 분포
Table 1. The distributions of entries in each groups.

그룹 / 데이터	그룹1 (8~15)	그룹2 (16~19)	그룹3 (20~23)	그룹4 (24~32)
Funet	0.99%	20.75%	17.58%	60.67%
MaeEast1	0.93%	19.85%	19.12%	60.10%
MaeEast2	0.98%	21.21%	20.07%	57.74%
MaeWest1	0.90%	20.30%	25.91%	52.89%
PacBell	0.71%	19.84%	17.99%	61.46%
Aads	0.71%	20.21%	18.49%	60.59%
MaeWest3	1.22%	25.68%	20.17%	52.92%
MaeWest2	1.26%	23.16%	17.96%	57.62%
평균	0.96%	21.37%	19.66%	58.00%

트리 개수에 대하여 충돌에 대비한 메모리의 여유를 얼마나 할당하는지를 의미한다.

표 1을 보면 여러 라우팅 데이터의 프리픽스 길이별 분포가 그룹별로 일정한 비율인 것을 확인할 수 있다. 그룹 1은 거의 약 1%정도인데 비해 그룹 2, 3은 약 20%이며 그룹 4는 가장 많은 60% 정도이다. 그러므로 기존 연구에서와 같이 모든 프리픽스 길이 또는 모든 그룹에 대하여 일정한 개수의 해쉬함수를 할당하는 것은 비효율적이다. 복수 해싱을 사용함에 따라 테이블 수와 충돌 발생 가능성의 trade-off관계가 있으므로 각 그룹별로 엔트리 수에 따라 적용하는 해쉬함수의 수를 가변 적용하면 적정한 테이블 수를 가지고 효과적으로 충돌 발생 가능성도 줄일 수 있다.

제안하는 구조에서는 표 1의 엔트리 비율 분석에 따라 그룹 1에는 1개의 해쉬함수를, 그룹 2와 3에는 2개의 해쉬함수를, 가장 많은 엔트리를 갖는 그룹 4에 대해서는 3개의 해쉬함수를 적용한다. 이에 따라 포워딩 테이블은 4개의 그룹에 대해 2개의 해쉬함수를 적용한 경우와 같은 8개의 테이블로 구현할 수 있다.

CRC-32 해싱 하드웨어로 적응적으로 적용된 복수의 해쉬 인덱스 값을 얻는 방식은 다음과 같다. 목적지 IP가 시스템에 들어오면 매 cycle마다 상위 bit부터 한 bit씩 CRC-32 해싱 하드웨어에 입력한다. 해당 길이만큼 cycle후에 CRC 레지스터에서 원하는 bit만큼을 인덱스 값으로 뽑아낼 수 있으며 시간차를 두고 다른 부분의 CRC 레지스터를 적절히 선택하여 복수의 해싱 인덱스 값을 동시에 뽑아낼 수 있다. 이에 따라 8 cycle이 지난 후에는 그룹1에 대한 해쉬 인덱스 1개를 추출하고 16, 20 cycle 후에는 각각 그룹 2, 3에 해당하는 해쉬 인덱스를 2개 추출하고 24 cycle 후에는 그룹 4에 해당하는 인덱스를 3개 추출한다.

3. 포워딩 테이블 구성을 위한 메모리 구조

가. 주 포워딩 테이블의 구성

구성 및 검색을 위한 메모리 접근은 프리픽스 입력에 대한 해쉬 인덱스 값을 메모리 주소로 일정하게 맵핑하여 이루어진다. 해쉬 인덱스에 의해 접근되는 메모리의 단위 블록을 버켓이라 한다. 하나의 프리픽스와 그와 관련된 정보를 저장하는 단위 블록을 로드(load)라 하며 각 버켓에는 복수 개의 로드(load)가 있다. 하나의 로드는 그림 2와 같이 프리픽스의 길이와 프리픽스, 그에 해당되는 아웃 포트 포인터(Out Port Pointer)를 저장하기 위한 필드로 구성된다.

프리픽스 길이는 8 ~ 32 bits 이기 때문에 그대로 저장한다면 32 bits를 표현하기 위한 5 bits가 필요하지만 그룹에 따라 처음 프리픽스 길이를 알고 있기 때문에 그룹 당 표현해야 할 프리픽스의 길이 개수만큼만 저장하면 된다. 제안하는 구조에서 그룹별로 표현해야 하는 프리픽스의 길이는 각각 8, 4, 4, 8개 이다. 그룹 4는 24 ~ 32 bit로 9가지 경우이지만 실제로 31bit 프리픽스는 존재하지 않으므로 8가지로 표현할 수 있다. 따라서 프리픽스 길이를 나타내기 위한 필드는 3 bits로 충분하다.

각 그룹별로 별도의 메모리 모듈에 테이블을 구성하기 때문에 하나의 메모리 모듈의 폭(width)은 동일해야 하므로 프리픽스를 저장하기 위한 공간은 가장 긴 프리픽스 길이를 기준으로 한다. 따라서 각 그룹별로 각각 15, 19, 23, 32bit를 저장할 수 있는 공간을 프리픽스 필드로 할당한다. 그러나 메모리 모듈의 접근을 위한 데이터 버스는 일정한 바이트 단위로 이루어지므로 접근되는 메모리의 폭(width)도 바이트 단위로 확장되어야 한다. 기존 연구에서는 버켓 단위로 읽고 쓰기를 하여 검색 및 구성 동작을 하였기 때문에 버켓의 크기를 바이트 단위로 확장하였다. 그러나 제안하는 구조에서는 더 효율적이고 빠른 구성 동작을 하기 위해 로드의 크기를 바이트 단위로 확장한다. 결과적으로 각 테이블의 로드의 메모리 폭은 그룹별로 각각 3, 4, 4, 5 Bytes가 된다.

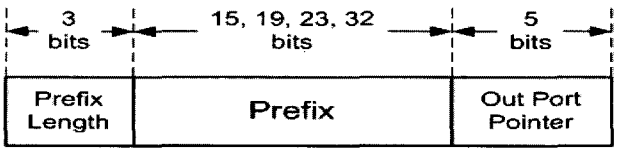


그림 2. 제안하는 구조에서의 한 로드의 구조
Fig. 2. The organization of a load in the proposed scheme.

버켓 당 로드 수가 아주 많다면 오버플로우 없이 충돌을 충분히 지원할 수 있지만 해쉬에 의해서 분산되는 이점에 비해 메모리 효율이 크게 떨어진다. 만일 똑같은 메모리 효율이라면 로드 수의 증가는 결국 해쉬 인덱스 비트 수가 줄어드는 것을 의미하므로 오히려 충돌을 많이 발생하게 될 것이다. 반면에 버켓 당 로드 수가 아주 적다면 각 버켓 당 충돌의 영향이 민감해져서 버켓이 가득차는 오버플로우가 많이 일어날 것이다.

그러므로 본 연구에서는 표 1의 그룹별 엔트리 분포를 분석하여 로드 수도 그룹별로 차등적으로 적용하였다. 같은 메모리 효율을 가정할 때, 그룹 1은 전체 엔트리 수가 적기 때문에 적용하는 해쉬함수가 적은 대신 로드 수를 많이 하는 것이 더 효율적이고 반대로, 그룹 4는 전체 엔트리 수가 많기 때문에 복수의 해쉬에 의해 충분히 엔트리가 분산될 수 있으므로 적용하는 해쉬함수의 수를 늘리고 로드 수를 더 작게 하는 것이 효율적이다. 즉 적용하는 해쉬함수가 많으면 그만큼 로드 수는 적게 할당하여 전체적인 메모리 효율에는 변화가 없도록 하였다. 결과적으로 버켓 당 그룹 1은 3개, 그룹 2와 3은 2개, 그룹 4는 1개의 로드를 할당하였다.

해쉬 인덱스 비트 수는 다음과 같이 결정하였다. 각 그룹 x별로 저장해야 할 전체 프리픽스 개수가 N_x, 테이블의 수를 T_x, 테이블 당 버켓의 수를 B_x, 버켓 당 로드의 수를 L_x이라 하면, 메모리 효율은

$$\text{메모리 효율} = \frac{N_x}{B_x \cdot L_x \cdot T_x} \quad (1)$$

로 나타낼 수 있다. 앞에서 언급한 바와 같이 각 그룹별로 해쉬함수 개수와 로드 수를 적응적으로 적용하여 결정하였으므로

$$T_1 = 1, \quad T_2 = 2, \quad T_3 = 2, \quad T_4 = 3$$
$$L_1 = 3, \quad L_2 = 2, \quad L_3 = 2, \quad L_4 = 1$$

이다. 따라서 메모리 효율을 1/3로 가정한다면, 각 그룹별 테이블 당 버켓의 수 B_x는

$$B_1 = 1, \; B_2 = \frac{3}{4}, \; B_3 = \frac{3}{4}, \; B_4 = 1,$$

가 된다. 해쉬 인덱스 비트 수, H_x는 2의 제곱으로 버켓의 수를 나타내므로

$$H_x = \lceil \log_2 B_x \rceil \quad (2)$$

표 2. 그룹별 테이블 수, 로드 수, 버켓 수
Table 2. The number of tables, loads and buckets in each groups.

	테이블 수	로드 수	버켓 수
그룹 1	1	3	N_1
그룹 2	2	2	$\frac{3}{4} N_2$
그룹 3	2	2	$\frac{3}{4} N_3$
그룹 4	3	1	N_4

로 결정된다.

이와 같이 적응적으로 적용된 해쉬함수에 따른 각 그룹별 테이블 수와 로드 수, 버켓의 수를 유도하여 주 포워딩 테이블을 구성하였다. 이를 정리하면 표 2와 같으며 이를 도식화하면 그림 3과 같다. 여기서 하나의 블록은 하나의 로드를 의미하며 N은 각 그룹의 엔트리 수를 의미한다.

나. 오버플로우 테이블의 구성

구성 동작 중에 해쉬 인덱스로 참조한 각 테이블의 버켓이 모두 가득 차 있어서 저장할 수 여유 로드가 없는 상태를 오버플로우라 정의한다. 이러한 경우 해당 엔트리는 작은 TCAM으로 구현한 오버플로우 테이블에 저장한다. TCAM은 다른 메모리에 비해 단가가 높고 상대적으로 저장용량이 적으며 전력소모가 크기 때문에 많은 크기의 TCAM를 사용하는 것은 주소 검색 구조에서 큰 부담이 된다. 따라서 오버플로우 테이블이 커질수록 더 많은 크기의 TCAM이 필요하므로 최대한 오버플로우 테이블의 크기는 줄여야 한다.

4. PSAU (Prefix Searcher & Allocator Unit)

제안하는 검색구조에서 테이블 구성동작과 검색동작을 수행하는 주 처리 유닛을 PSAU (Prefix Searcher & Allocator Unit)라 정의한다. PSAU의 하드웨어 도식도는 그림 3과 같다.

PSAU의 수행 동작은 크게 구성동작과 검색동작으로 구분할 수 있으며 시스템 제어신호에 따라 두 가지 모드로 동작된다. 동작에 대한 상세한 설명은 뒤에서 더 자세히 하겠다. PSAU은 크게 입력신호 및 해쉬 처리부, Overflow 처리기, Priority Encoder 및 출력부와 여러 GSA(Group Searcher & Allocator)로 구성된다.

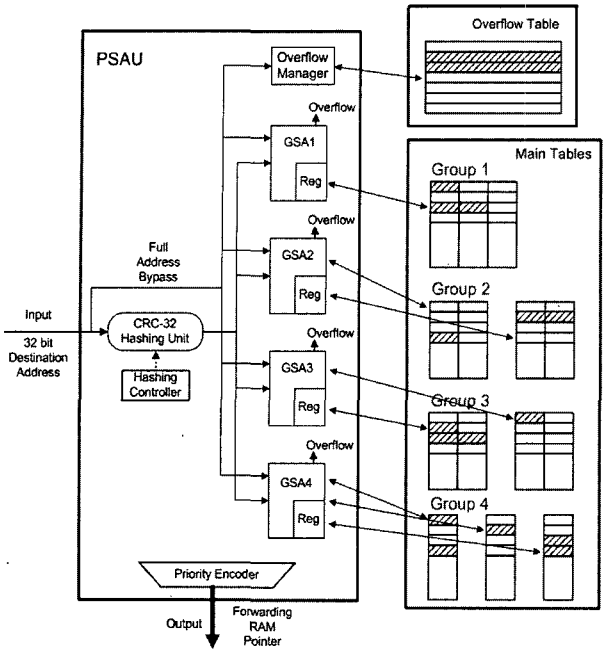


그림 3. 제안하는 IP 주소 검색 구조와 포워딩 테이블 구조

Fig. 3. Proposed IP address lookup scheme with the structure of forwarding tables.

가. 입력 및 해쉬 처리부

입력신호로는 각 패킷의 헤더 정보 중 32bit의 목적지 주소 전체를 받는다. 받은 주소 값은 두 패스로 나누어진다. 하나의 패스로는 버퍼링을 통해 한 cycle 당 상위 한 bit씩 CRC-32 해쉬 유닛에 입력된다. CRC-32 해쉬 유닛에서는 프리픽스 길이만큼 cycle이 경과한 후 복수의 해쉬 인덱스 값을 CRC 레지스터에서 선택하여 GSA로 출력한다. 다른 패스로는 주소 값 전체를 bypassing하여 GSA로 보낸다. 즉 메모리 참조를 위한 해쉬 인덱스 값과 구성 및 검색을 위한 원본 데이터를 각 GSA로 보내게 된다.

'Hashing Controller'는 각 그룹별로 인덱스 값 추출을 위한 CRC 레지스터의 선택과 CRC-32 해쉬 유닛에서 GSA로 보내는 해쉬 인덱스의 길이와 개수를 저장하고 있으며 제어할 수 있다. 이 3가지 설정치는 해당 네트워크 환경에 따라 더 효율적이도록 최적화될 수 있으며 가변적일 수도 있다. 따라서 'Hashing Controller'는 가변적으로 설정치를 조정할 수 있도록 관리자에 의해 세팅 가능토록 하였다. 또한 필요에 따라 라우터 또는 PSAU 내부에서 추가 유닛을 통해 입력되는 라우팅 데이터의 엔트리를 모니터링하여 분석함으로서 적응성 있게 자동적으로 설정치를 최적화할 수도 있다.

나. GSA (in each Groups, Searcher & Allocator)

각 그룹 별로 존재하는 GSA는 검색을 위한 비교 기능과 구성을 위한 메모리 할당 기능을 수행한다. 그룹별 GSA의 입출력 개수는 'Hashing Controller'에서 설정된 그룹별 해쉬 개수와 인덱스 길이에 따라 정해진다. 입력으로는 bypassing된 목적지 주소 값과 CRC-32 해쉬 유닛에서 받은 복수의 해쉬 인덱스 값을 받는다.

GSA 내부의 레지스터에는 테이블의 각 버켓 당 빈 로드의 수를 저장하고 있다. 기존의 연구에서는 각 버켓의 유효 로드 수를 메모리의 버켓 내 필드에 저장하여 전체 버켓의 내용을 모두 읽어 들인 다음에 구성 및 검색을 해야 하는 단점이 있었다. 특히 오버플로우 발생시에도 해당 테이블의 버켓 내용을 읽어 들여야 하기 때문에 비효율적이었다. 따라서 본 연구의 제안하는 구조에서는 각 버켓 당 유효 로드 수를 GSA의 내부 레지스터에 저장하여 테이블의 내용을 읽어 들이지 않고도 더 빠르게 검색 및 구성 동작을 할 수 있으며 오버플로우 발생 시 불필요한 동작을 줄일 수 있다.

GSA에서의 구성동작은 다음과 같다. 먼저 복수의 해쉬 인덱스 값을 입력받아 각 인덱스가 참조하는 버켓에 해당하는 내부 레지스터 값을 읽어서 버켓의 유효 로드 수를 비교한다. 그 중 더 많은 유효 로드 수를 갖고 있는 테이블의 버켓을 선택하여 할당한 후 bypassing된 프리픽스 데이터와 관련 정보를 쓰기 작업을 통해 해당 테이블의 버켓 내의 로드에 저장한다. 만약 해쉬 인덱스가 참조하는 모든 버켓에 유효 로드가 없는 꽉 찬 상태이면, 오버플로우 판정을 하여 오버플로우 신호를 오버플로우 처리기로 보낸다. 이와 같이 메모리 접근은 쓰기 작업 단 한번만이 필요하다.

GSA에서 검색동작은 다음과 같다. 구성 동작과 같이 복수의 해쉬 인덱스 값을 입력받아 각 인덱스가 참조하는 각 테이블의 버켓의 모든 데이터를 읽어 들인다. 모든 버켓의 데이터를 읽어 왔으므로 GSA 내부에서 각 로드 단위로 나누어 매치 작업을 수행한다. 매치 작업은 프리픽스 길이에 상관없이 모든 매치 결과를 찾는다. 그 모든 결과는 Priority Encoder로 보낸다. 따라서 검색동작 중 메모리 접근도 읽기 작업 단 한번만이 필요하다.

다. 오버플로우 처리기

오버플로우 처리기는 TCAM으로 구성된 오버플로우 테이블의 구성 및 관리를 담당하는 유닛이다. TCAM 기반이기 때문에 한번의 접근으로 바로 검색결과를 얻

을 수 있으며 추가 업데이트(incremental update)가 가능하도록 테이블을 관리를 위한 알고리즘을 수행한다.

오버플로우 처리기는 구성동작 중 GSA에서 오버플로우가 발생하여 오버플로우 신호를 전송하면 해당 주소 값을 주 포워딩 테이블이 아닌 오버플로우 테이블에 저장한다. 검색동작 중에는 GSA의 검색동작과 병렬적으로 오버플로우 테이블을 검색한다. 검색한 결과는 각 그룹별 GSA에서 매치된 결과와 같이 Priority Encoder로 보낸다.

라. Priority Encoder 및 출력부

Priority Encoder는 각 그룹별 GSA와 오버플로우 처리기에서 보내 온 모든 매치 결과 중에 가장 긴 프리픽스 길이를 갖는 결과 값을 선택한다. 즉 LPM를 수행한다. 최종 출력으로는 결정된 프리픽스에 해당하는 Forwarding RAM Pointer를 출력한다.

5. 포워딩 테이블 구성 동작

제안하는 구조의 포워딩 테이블 구성 동작을 정리하면 다음과 같다. 이처럼 어떠한 경우에도 단 한 번의 메모리 접근, 쓰기 동작을 통해서 포워딩 테이블을 구성할 수 있다.

1) PSAU로 프리픽스와 프리픽스 길이 정보, 그에 해당되는 아웃 포트 포인터(Out Port Pointer)를 입력한다.
2) 입력 받은 세 가지 정보를 프리픽스 길이에 따라 해당하는 그룹의 GSA로 bypassing하여 보낸다.
3) 입력된 프리픽스를 버퍼링을 통하여 상위 bit부터 한 cycle에 한 bit씩 CRC-32 해쉬 유닛에 입력한다. CRC-32에서는 프리픽스 길이만큼의 cycle 후에 해당 그룹의 해쉬 개수와 인덱스 길이만큼 해쉬 인덱스를 출력하여 해당 GSA로 보낸다.
4) 해당 그룹의 GSA에서는 단일 또는 복수의 해쉬 결과에 의해 참조될 버켓의 유효 로드 수 정보를 내부 레지스터에서 읽어 들여 비교한다. 이 중 빈 로드 수가 더 많은 버켓을 선택하여 프리픽스를 저장할 버켓과 로드를 결정한다. 만약 빈 로드 수가 많은 테이블이 여러 개이면 임의의 우선순위를 통해 결정한다.
5) 결정된 버켓과 로드 정보에 따라 메모리 쓰기 작업을 위한 주소 인코딩을 수행하고 할당된 로드에 2)에서 bypass된 세 가지 정보를 저장한다.
6) 만약 해당 버켓에 빈 로드가 없다면 오버플로우 처리기를 통해서 오버플로우 테이블에 저장한다.

6. 포워딩 테이블 검색 동작

제안하는 구조의 포워딩 테이블 검색 동작을 정리하면 다음과 같다. 이처럼 단 한 번의 메모리 접근, 읽기 동작을 통해서 주소 검색을 완료할 수 있다.

1) 라우터에 들어온 목적지 주소가 PSAU에 입력된다.
2) 입력된 목적지 주소 32 bits를 모든 GSA로 bypass하여 보낸다.
3) 입력된 목적지 주소를 버퍼링을 통하여 상위 bit부터 한 cycle에 한 bit씩 CRC-32 해쉬 유닛에 입력한다. CRC-32에서는 그룹별 처음 프리픽스 길이만큼의 cycle 후에 각 그룹별 해쉬 인덱스 값을 추출한다. 본 연구에서 제안한 적응적으로 적용한 해쉬 방식에 따라 각 8, 16, 20, 24 cycle 경과 후에 각각 그룹 1은 1개, 그룹 2와 3은 2개, 그룹 4는 3개의 해쉬 인덱스 값을 추출하여 각 그룹별 GSA로 보낸다.
4) 각 그룹별 GSA에서는 해쉬 인덱스 값이 참조하는 각 테이블의 버켓에 있는 모든 유효 로드의 정보를 읽는다. 읽어 들인 로드의 모든 정보 중에 매치되는 결과를 모두 Priority Encoder로 보낸다.
5) GSA에서의 검색 동작과 병렬적으로 오버플로우 테이블의 데이터 중에서도 검색작업을 한다. 이렇게 검색된 매치 정보 역시 Priority Encoder로 보낸다.
6) Priority Encoder에서는 GSA와 오버플로우 테이블에서 검색되어 매치된 모든 결과 중에서 프리픽스 길이가 가장 긴 엔트리를 선택한다. 선택된 엔트리의 Forwarding RAM Pointer 정보를 출력하여 주소 검색을 완료한다.

Ⅳ. 실험 및 성능평가

본 연구에서는 분포 분석에서 이용했던 바와 같이 8개의 실제 백본 라우터의 라우팅 데이터를 이용하여 시뮬레이션을 수행하였다. 실험은 기존의 그룹별로 일정 복수 해쉬을 적용한 방식과 제안하는 적응적으로 적용한 복수 해쉬을 이용한 방식을 비교 분석하기 위해 동일한 메모리 효율과 동일한 테이블 수를 구현하고 두 경우에서 충돌(collision)의 집중으로 인해 발생하는 오

버플로우의 수를 비교하였다.

오버플로우 수의 감소는 충돌을 효과적으로 분산하고 제어하였다는 것을 의미한다. 더욱이 오버플로우 테이블 구성을 위한 TCAM의 크기를 최소한으로 줄임으로서 TCAM의 엔트리가 많아짐에 따라 발생하는 단점을 완화할 수 있다.

메모리 효율은 포워딩 테이블을 위해 할당된 전체 메모리 크기에서 테이블 구성을 위해 사용되고 있는 메모리의 비율을 의미하므로 메모리 효율을 높일 수 있다는 것은 필요한 메모리 크기를 줄일 수 있다는 것을 의미한다. 그러나 앞서 언급한 바와 같이 메모리 크기를 줄이는 것은 충돌 및 오버플로우 발생 비율과 trade-off 관계가 있으므로 메모리 크기를 최소화하면서도 오버플로우를 줄일 수 있을 때 그 검색 구조는 효율적이라 할 수 있다. 또한 네트워크 환경과 시간에 따라 각 라우팅 데이터의 엔트리는 가변적이지만 표 1의 분석 결과와 같이 각 그룹별 프리픽스는 일정한 비율을 유지하므로 제안하는 그룹별 해쉬함수의 개수와 로드 수, 메모리 효율은 합리적이었다.

일정한 메모리 효율을 전제하여 비교하기 위해 본 실험에서는 메모리 효율을 1/3로 균일화 하였다. 즉 N 개의 프리픽스를 저장하기 위해 전체 로드 수는 $3N$이다.

이에 따라 각 그룹별 해쉬 인덱스의 길이에 의한 버켓의 수를 결정하였으며 실험상에서는 메모리의 절대 크기가 포워딩 테이블을 위한 메모리 구성, 즉 버켓 또는 로드의 구조에 따라 달라지므로 정확히 1/3의 메모리 효율로 하기는 힘들었다. 그래서 가장 1/3의 메모리 효율에 가깝게 구현하였으며 비교 대상인 두 개의 해쉬 함수를 그룹별로 균일하게 적용한 경우 보다는 더 작은 메모리 크기를 선택하였다.

이와 같이 동일한 전제 조건을 갖추고 각 라우팅 데이터 별로 균일하게 단일 해쉬함수를 적용하는 경우와 두 개의 복수 해쉬함수를 적용하는 경우에 대하여 제안하는 방식을 비교, 실험하였다. 표 3은 그 결과를 정리한 것이다. 성능 비교지표로서 메모리 크기와 오버플로우 수를 비교하였다. 또한 오버플로우 수는 메모리 효율과 구성 및 검색 구조의 효율성 외에도 테이블의 수에 의존하기 때문에 각 실험에 대한 테이블 수를 명시하였다. 표 3에서 알 수 있듯이 테이블 수가 적은 단일 해쉬을 이용한 경우에는 그룹화를 통해 테이블의 수는 줄였지만 오버플로우 수가 매우 많아진 것을 알 수 있다. 이에 반해 균일한 복수 해쉬을 적용한 경우와 제안하는 구조 모두 다 테이블 수가 8개로 늘어남에 따라 오버플로우 수가 크게 줄어든 것을 볼 수 있다.

제안하는 구조를 테이블 수가 동일한 균일한 두 개의 해쉬을 적용 경우와 비교해보면, 전체적으로 비슷한 메모리 효율에서 대하여 메모리 오버플로우 수가 크게 줄었음을 알 수 있다. 특히 엔트리 수가 적은 Aads와 MaeWest2 라우팅 데이터에 대해서 보다 엔트리 수가 많은 Funet, MaeEast1 등에서 더 좋은 결과를 보였다.

엔트리 수가 적은 데이터에서 기대해 미치지 못한 결과가 나온 이유는 엔트리 수가 적으면 해쉬의 분산 효과에 의해서 보다 로드 수 증가에 따른 충돌 가능성 감소 효과가 더 크기 때문이다. 이는 앞서 적응적 적용에서 설명하였던 그룹 1이 그룹 4보다 엔트리 수가 매우 적기 때문에 해쉬함수 수를 줄이고 로드 수를 늘였던

표 3. 라우팅 데이터 별 비교실험 결과
Table 3. Simulation results with various routing data.

라우팅 데이터	엔트리 수	기존의 균일한 해쉬 적용[8] (One or Two Hashing)			제안하는 검색구조 (Adaptive Multiple Hashing) 테이블 수=8	
		메모리크기(KB)	오버플로우 수		메모리크기(KB)	오버플로우 수
			1 Hashing 테이블 수=4	2 Hashing 테이블 수=8		
Funet	41584	724	1733	389	612	55
MaeEast1	39902	724	612	41	676	18
MaeEast2	38470	462	4541	37	676	23
MaeWest1	29584	459	1946	414	370	46
PacBell	20637	361	342	33	337	18
Aads	20328	361	345	10	369	8
MaeWest3	15050	231	291	1	217	23
MaeWest2	14636	205	379	1	186	21

것과 같은 이치이다.

결국 그룹 4의 로드가 하나인 제안한 방식은 엔트리 수가 적은 데이터에서는 큰 효과를 얻지 못했다. 제안한 방식이 복수 해싱의 효과를 극대화 하면서도 테이블 수를 줄이기 위한 방법이고 버켓의 수와 로드 수는 적절히 선택하면 메모리 효율을 높일 수 있으므로 이와 같은 경우에는 로드 수를 늘리고 해쉬 인덱스 길이를 줄이는 방법이 더 효과적일 것으로 판단된다. 이를 바탕으로 MaeWest2 데이터에 대하여 그룹 4의 테이블을 로드 수를 2로 늘리고 해쉬 인덱스 길이를 1 bit줄여서 동일한 메모리 크기에서 추가 시뮬레이션을 수행하였다. 그 결과, 그룹 4에서는 오버플로우가 발생하지 않았다.

표 4는 본 논문에서 제안하는 구조와 기존의 다른 알고리즘들의 검색 성능 및 필요한 메모리 크기를 비교하였다. 각 알고리즘은 약 30,000개의 엔트리를 갖는 MaeWest1 라우팅 데이터를 이용하여 비교실험하였다. 이 결과에서 알 수 있듯이 제안하는 구조는 모든 경우에 대해서 검색과정 시 메모리 접근이 단 한번 만에 가능하므로 다른 알고리즘에 비해 더 고속으로 검색을 수행할 수 있다. 또한 같은 검색 성능을 가지고 있는 TCAM기반 방식과 균일한 복수 해싱을 적용한 방식에 비해 필요한 SRAM과 TCAM의 크기가 매우 작으므로 메모리 효율 측면에서도 매우 우수하다. 특히 기존 연구 중 SFT를 제외하고는 가장 작은 메모리 크기에 포워딩 테이블을 구성할 수 있었다.

표 5는 기존의 해쉬함수 기반의 주소 검색 구조와 제안하는 구조의 구성동작 시 메모리 접근 회수를 비교한

표 4. 기존 알고리즘과 성능 비교
Table 4. Comparison with existing lookup schemes.

주소 검색 구조	검색동작 중 메모리 접근 회수 (최소 / 최대)	필요한 SRAM 크기 (Kbytes)	필요한 TCAM 크기 (# of entries)
단일 해싱 적용[4]	1 / 5	189	0
복수해싱 및 그룹화[8]	1 / 1	459	414
Range search[9]	1 / 16	376	0
SFT[10]	2 / 9	150 ~ 160	0
DIR-24-8[11]	1 / 2	33,000	0
DIR-21-3-8[11]	1 / 3	9,000	0
Huang's[12]	1 / 3	450 ~ 470	0
TCAM-based[13]	1 / 1	0	30,000
제안하는 구조	1 / 1	370	46

표 5. 기존 해싱 기반 알고리즘과 구성성능 비교
Table 5. Comparison about the building with existing lookup schemes using hashing.

주소 검색 구조	구성동작 중 메모리 접근 회수 (최소 / 최대)
단일 해싱 적용[4]	2 / 2
복수 해싱 및 그룹화[8]	2 / 2
제안하는 구조	1 / 1

것이다. 기존의 알고리즘은 버켓 단위로 접근이 이루어지므로 구성동작 시 해당 버켓의 데이터를 일단 읽은 다음 처리하고 다시 테이블에 쓰는 동작이 필요하였다. 반면 제안하는 구조는 각 버켓의 유효 로드 수를 레지스터에 저장하고 있고 로드 단위로 메모리에 접근이 가능하기 때문에 바로 쓰기 동작을 통해 해당 로드에 구성할 수 있다. 따라서 제안하는 구조는 모든 경우에 한 번의 메모리 접근으로 테이블을 구성할 수 있다.

이상의 실험 결과에서와 같이 본 논문에서 제안하는 복수 해싱의 적응적인 적용에 의한 구성 및 검색 구조는 기존의 균일한 복수 해싱을 적용한 방법에 비해 메모리 효율이 우수하며 더 효율적인 충돌의 분산으로 오버플로우를 줄일 수 있었다. 또한 한 번의 메모리 접근으로 빠르게 포워딩 테이블의 구성 및 검색이 가능하며 시스템 내장 가능한 매우 작은 메모리로 구현 가능한 효율적인 구조임을 확인할 수 있었다.

V. 결 론

본 논문에서는 라우팅 데이터의 엔트리 분포 분석을 바탕으로 프리픽스를 길이별로 그룹화하고 그룹별로 해쉬함수의 수를 적응적으로 적용하여 충돌(collision)의 발생 가능성을 적절한 개수의 메모리 모듈에서도 충분히 감소시켜 메모리 구조가 효율적인 IP 주소 검색 구조를 제안하였다. 또한 메모리의 버켓 구조를 개선하여 기존 보다 더 적은 bits로 프리픽스 정보를 저장할 수 있었으며 한 번의 메모리 접근으로 빠른 포워딩 테이블 구성 및 검색이 가능하도록 하였다. 마지막으로 이러한 동작을 효율적으로 할 수 있는 하드웨어 구조, PSAU(Prefix Searcher & Allocator Unit)를 제안하였다. 이를 실험을 통하여 동일한 메모리 효율과 테이블 수를 전제 조건으로 그룹별로 균일하게 적용된 복수 해싱방식 보다 제안하는 가중 적용된 복수 해싱 방식이 효율적임을 보였다.

따라서 제안하는 IP 주소 검색 구조는 초고속 네트워킹을 위한 스위치 및 라우터에서 포워딩 테이블의 빠른 구성 및 검색이 가능하며 확장성이 우수하고 시스템에 내장 가능한 매우 작은 SRAM과 TCAM으로 구현할 수 있는 메모리 효율이 높은 구조이다.

참 고 문 헌

[1] X. Sun and Y. Q. Zhao, "An on-chip IP address lookup algorithm," *IEEE Transactions on Computers*, pp.873-885, July 2005.

[2] K. McLaughlin, S. O'Kane and S. Sezer, "Implementing High Speed IP Address Lookups in Hardware," *Proc. AICT/SAPIR/ELETE'05*, pp.140-144, July 2005.

[3] V. Fuller et al., "Classless Inter-Domain Routing(CIDR): An address assignment and aggregation strategy," *RFC 1519*, June 1993.

[4] H. Lim, J. Seo and Y. Jung, "High Speed IP Address Lookup Architecture Using Hashing," *IEEE Communications Letters*, Vol.7, No.10, pp.502-504, Oct. 2003.

[5] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *Proc. of ACM Sigmetrics'98*, pp.111, 1998.

[6] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," *Proc. IEEE INFOCOM'01*, pp.1454-1463, 2001.

[7] H. Lim and Y. Jung,"A Parallel Multiple Hashing Architecture for IP Address Lookup," *Proc. IEEE HPSR'04*, pp. 91-95, 2004.

[8] 김혜란, 정여진, 임창훈, 임혜숙, "프리픽스그룹화를 이용한 병렬 복수해싱 IP 주소검색구조," *한국통신학회 논문지*, Vol .30, No.3B, pp.65-72, Mar. 2005.

[9] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," *Proc. IEEE INFOCOM'98*, pp. 1248-1256, Apr. 1998.

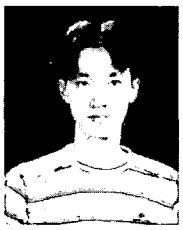
[10] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," *in Proc. ACM SIGCOMM'97*, pp.3-14, Sept. 1997.

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

[12] 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.

[13] A. McAuley and P. Francis, "Fast Routing Table Lookup Using CAMs," *Proc. IEEE INFOCOM'93*, pp.1382-1391, March 1993.

저 자 소 개



박 현 태(학생회원)
2004년 연세대학교 전기전자공학과 학사 졸업
2006년 연세대학교 전기전자공학과 석사 졸업
2006년 현재 연세대학교 전기전자공학과 박사 과정
<주관심분야 : 고속네트워크 관련 SoC설계 및 네트워크 프로세서 설계>

문 병 인(정회원)
1995년 연세대학교 전자공학과 학사 졸업
1997년 연세대학교 전자공학과 석사 졸업
2002년 연세대학교 전자공학과 박사 졸업
2004년 하이닉스반도체 선임연구원
2005년 연세대학교 연구교수
2006년 현재 경북대학교 전자전기컴퓨터학부 전임강사
<주관심분야 : 디지털 집적회로 설계, SoC 설계, 마이크로프로세서 구조 설계>

강 성 호(평생회원)
1986년 서울대학교 제어계측공학과 학사 졸업
1988년 The University of Texas, Austin 전기 및 컴퓨터공학과 석사 졸업
1992년 The University of Texas, Austin 전기 및 컴퓨터공학과 박사 졸업
1992년 미국 Schlumberger Inc. 연구원
1994년 Motorola Inc. 선임 연구원
2006년 현재 연세대학교 전기전자공학과 교수
<주관심분야 : SoC 설계 및 응용, DFT, SoC 테스트>