능동적 트라이 압축을 이용한 고속 IP 검색

오 승 현†

요 약

라우터의 IP 주소검색은 라우터에 도착한 IP 패킷의 목적지 주소를 이용하여 적절한 출력링크를 검색하고 결정하는 것이다. IP 주소검색은 라우터 성능의 병목지점 중의 하나로써 고속 백본망에 필요한 초고속 라우터 개발에 필수적인 부분이다. 본 논문은 보통의 펜티엄 CPU에서 능동적인 트라이(Trie) 압축기법을 이용하여 작은 메모리만으로 기가비트급 IP 주소검색을 실시할 수 있는 동적 트라이 압축(Dynamic Trie Compression) 자료구조를 소개한다. DTC 자료구조는 트라이를 압축하여 포워딩 테이블을 만들 때 테이블의 크기와 검색속도의 상관관계를 고려하여 능동적으로 테이블의 크기를 선택할 수 있다. 또한 트라이를 압축할 때 트라이의 구조를 반영하여 자료구조의 크기를 최소화함으로써 포워딩 테이블에 대한 IP 주소검색이 고속의 SRAM 캐시 검색이 되도록 한다. 실험결과에서 DTC 자료구조는 다양한 라우팅 테이블에 대해 능동적으로 최적의 압축을 제공함으로써 보통의 펜티엄 CPU에서 최대 12.5×10^6 LPS(Lookup per second)를 기록하였다.

A Fast IP Lookups using Dynamic Trie Compression

Seung Hyun Oh†

ABSTRACT

IP address lookup of router searches and decide proper output link using destination address of IP packet that arrive into router. The IP address lookup is essential part in the development of high-speed router needed to high-speed backbone network as one of bottleneck of router performance. This paper introduces DTC data structure that can support gigabit IP address lookup by dynamic trie compression technique that just uses small memory in conventional Pentium CPU. When make a forwarding table by trie compression, the DTC can dynamically select a size of data structure with considering correlation between table's size and searching speed. Also, when compress the prefix trie, DTC makes IP address lookup on the forwarding table of a search on the high speed SRAM cache by minimizing the size of data structure reflecting the structure of the trie. In the experiment results, the DTC data structure recorded performance of maximum 12.5×10^6 LPS (lookup per second) in conventional Pentium CPU through a dynamic building of most suitable compression over variety of routing tables.

키워드 : 라우팅(Routing), 포워딩 테이블(Forwarding Table), 프리픽스(Prefix), 트라이(Trie), 주소검색(Address Lookup), 다음-홉(Next- Hop)

1. 서 론

근래의 인터넷은 초고속 액세스망의 일반적인 보급과 인터넷 접속자의 급격한 증가 때문에 발생한 트래픽의 증가로 백본망의 초고속화가 진행되고 있다. 백본망의 확장은 네트워크 케이블의 증설과 라우터 처리속도 증가로 대표된다. 네트워크 케이블의 용량은 지속적으로 증가하여 테라비트로 전이되고 있는 상태이다. 그러나 라우터의 처리속도는 그에 미치지 못하여 일반적으로 기가비트급 처리속도를 보이고 있고 최근 들어 테라비트급 라우터가 시험적으로 소개되고 있다. 일반적으로 라우터는 세 가지 기능을 가지고 있다. 첫째, 라우터에 도착한 패킷의 처리순서를 결정하는 scheduling 기능이다. 둘째, 패킷을 출력단자(output port)로 전달하는 스위칭(Switching)이며, 마지막으로 세 번째 기능은 패킷의 목적지 IP 주소를 바탕으로 다음-홉(Next- hop) 주소를 결정하는 IP 주소 검색기능이다. IP 주소검색 기능은 CIDR[1] 주소체계를 지원하는 BGP-4[2] 라우팅 프로토콜의 보급에 따라 supernet 개념이 도입됨으로써 최장일치 검색(LPM : Longest prefix matching)을 지원해야 됨에 따라 O(1)의 속도를 달성하기 어려운 문제가 되었다.

IP 주소검색의 연구는 크게 하드웨어를 이용하는 방법[3, 4]과 소프트웨어를 기반으로 하는 방법[5, 6]으로 구분할 수 있다. 하드웨어를 이용하는 방법은 CAM(Content Access Memory)을 이용하는 방법[3], 주소검색용 하드웨어 로직을 구성하는 방법[4] 등이 있다. 소프트웨어를 이용하는 방법은 포워딩 테이블의 구조를 고속검색이 가능하도록 변경함으로써 IP 주소검색의 성능을 향상시키는 방법이다. 또한 최장일치검색을 일반적인 이진검색으로 변경[16]하거나 MPLS [9]와 같이 프로토콜을 변경하는 연구도 소프트웨어를 이용하는 범위에 속한다. 자료구조를 변경하는 연구의 경우 해

※ 본 연구는 2002년도 동국대학교 신임교원연구비지원으로 이루어졌음.
† 정 회 원 : 동국대학교 컴퓨터학과 교수
논문접수 : 2003년 7월 19일, 심사완료 : 2003년 10월 9일

시 테이블과 트라이 구조 등이 일반적으로 사용되었다. 참고로, 트라이는 사전과 같이 단어를 순서대로 나열하여 자료를 검색하는 형태에 적용되는 자료구조로서 대부분의 단어들의 프리픽스가 공유된다는 점을 이용하는 트리와 유사한 구조이다. 라우팅 테이블을 트라이로 표현할 때는 IP 주소의 각 비트가 트라이의 링크로 사용되므로 링크의 집합으로 표시되는 경로는 하나의 리프노드(Leaf node)로 연결되고, 리프노드는 경로에 해당하는 IP 주소에 대한 라우팅 정보를 갖고 있다. 따라서 IP 주소검색은 IP 주소의 각 비트를 이용하여 링크를 탐색하고 마지막에 도착한 리프노드에 담긴 정보가 라우팅 정보로 선택된다.

포워딩 테이블의 자료구조를 변경하여 고속 IP 주소검색을 하는 연구들은 대부분 다음의 두 가지 개념으로 정리할 수 있다. 포워딩 테이블 구축에 필요한 메모리의 양이 커지는 대신 검색횟수를 감소[5] 시키거나 메모리의 양을 L2 캐시 크기보다 작게 함으로써 메모리 검색을 캐시 검색으로 대치[6]하는 것이다. 두 가지 개념을 트라이에 적용하면, 첫 번째 개념의 경우 트라이의 리프노드에서 라우팅 정보를 수집하여 메모리에 저장하면 IPv4 주소에 대해 2^{32}개의 엔트리를 갖는 해시 테이블을 메모리에 저장하고 1회의 메모리 검색으로 IP 주소검색을 완료할 수 있다. 그러나 이때 필요한 메모리의 크기가 한 엔트리를 한 비트로 표시하여도 약 4GB 이상으로 현실적으로 적용하기 어려우므로 메모리의 크기를 적당량 줄이면서 검색횟수를 감소시키는 방법이 필요하다. 이러한 현실적인 어려움은 저가의 펜티엄 PC에서 소프트웨어만으로 고속의 라우터를 구현하려는 환경에서 기인한다. 두 번째 개념 즉, 자료구조의 크기를 L2 캐시보다 작게 하여 속도가 빠른 캐시 검색 효과를 얻으려면 트라이 구조와 라우팅 정보에 다양한 자료구조를 적용하여 메모리 크기를 압축한다. 캐시보다 작은 크기의 포워딩 테이블을 캐시에 저장하면 메모리 검색을 캐시 검색으로 대치할 수 있으므로 검색횟수가 증가하더라도 더 빠른 검색속도를 얻을 수 있다. 본 논문은 2-단계 트라이 구조로 트라이의 단계 분할 깊이에 따라 변화하는 자료구조의 크기와 IP 주소 검색속도의 상관관계를 밝히고, 라우팅 테이블의 크기와 종류에 상관없이 일정한 검색속도를 갖는 포워딩 테이블 구조와 검색 알고리즘을 제시한다. 또한 트라이 구조를 압축할 때 동적으로 트라이의 깊이를 측정하여 자료구조의 크기를 최소화한다.

본 논문의 포워딩 테이블 자료구조는 트라이를 상하 두 개로 분리하여 포워딩 자료구조를 만드는 2-단계 트라이 구조를 사용한다. 트라이의 적절한 분할 개수와 깊이를 동적 프로그래밍 기법을 이용하여 결정하는 연구[14]에서는 2-단계부터 6-단계까지 다양한 단계를 대상으로 실험하였다. 그 결과 400KB부터 9.8MB까지의 자료구조 크기와 200ns ~700ns까지의 검색속도의 분포를 보고하였다. 이때 2-단계 트라이에서 200ns의 가장 빠른 검색속도와 9.8MB의 가장 큰 크기를 기록하였다. 본 논문은 [14]의 단계별 트라이 검색속도와 자료구조의 크기에 주목하여 트라이를 비트맵으로 변환할 때 가장 속도가 빠른 2-단계 구조를 선택하고 자료구조의 크기가 커지지 않도록 세 가지 방법을 이용하여 자료구조를 설계하였다. 첫째, 프리픽스 노드정보와 차일드 링크정보를 최소의 크기로 저장하기 위해서 비트맵을 사용한다. 두 번째, 2-단계 트라이의 분할 깊이를 트라이의 구성상태 즉, 라우팅 테이블의 상태를 반영하여 자료구조의 크기가 최소가 되는 최적의 깊이를 동적으로 선택하도록 하였다. 마지막으로 라우팅 테이블의 프리픽스 길이가 다양하게 분포되는 점을 반영하여 하위 단계의 트라이의 깊이가 변화될 수 있도록 하였다.

논문의 나머지는 다음과 같은 순서로 구성되어있다. 2장에서는 기존연구에 대해 살펴보고, 3장에서는 본 논문에서 제시한 자료구조와 관련 검색 알고리즘의 상세한 구조를 기술한다. 4장에서는 실험결과를 제공하며 마지막으로 5장에서는 결론과 향후 연구방향에 대해 기술한다.

2. 기존 연구

라우터 성능의 마지막 병목지점으로 알려진 IP 주소검색의 성능은 CPU의 계산속도보다 메모리 검색속도가 월등히 느리기 때문에 계산량보다 메모리 검색횟수와 메모리의 속도에 따라 결정된다. 따라서 IP 주소검색 연구를 메모리 검색횟수를 기준으로 분류하면 상대적으로 빠른 메모리를 이용하려는 연구와 메모리 검색횟수를 감축하려는 부류로 나눌 수 있다. 빠른 메모리를 이용하려는 연구는 고속의 하드웨어를 사용하여 메모리 검색속도를 빠르게 하려는 연구들[8]과 IP 주소검색 과정을 생략함으로써 라우터의 성능을 개선하려는 연구로 분류할 수 있다. 내용기반검색(Content Access Memory)을 이용한 방법[3], 주소검색을 위한 ASIC 디자인을 제안하는 연구[4]들이 빠른 메모리를 사용하는 경우이다. IP 주소검색 과정을 생략하려는 연구는 태그(tag)를 사용하여 스위칭을 하는 MPLS[9]가 대표적인 경우이다.

메모리 검색횟수를 감축하는 방법들은 대체로 소프트웨어 기반의 연구들이다. 소프트웨어 기반의 연구는 주소검색에 특별한 하드웨어적 지원을 배제하고 주소검색을 빠르게 진행할 수 있는 포워딩 테이블 자료구조를 고안하는데 집중하고 있다. 이러한 부류의 연구들은 크게 두 가지 방법으로 구분 할 수 있다. 충분한 메모리를 사용하여 자료구조를 만듦으로써 메모리 검색을 최소화하려는 부류와 자료구조의 크기를 작게 함으로써 빠른 속도의 캐시 메모리를 사용하려는 부류가 그것이다.

작은 크기의 자료구조를 이용하여 메모리 압축을 시도하는 IP 검색 알고리즘에 많이 사용된 자료구조로는 래딕스

트리(radix tree)[10]와 이진 트라이[11]를 들 수 있다. 이진 트라이는 루트부터 리프노드 혹은 중간노드까지의 경로가 하나의 값을 의미하는 간단한 자료구조이다. 즉, 경로를 구성하는 0과 1의 비트열과 일치하는 프리픽스의 라우팅 정보가 경로의 끝에 달려있는 노드에 저장된다. 패트리샤 트라이(Patricia trie)[12]는 이진 트라이를 개량한 것으로 트라이의 레벨을 압축하고 불필요한 중간노드를 생략하여 메모리를 압축한다. 패트리샤 트라이는 BSD 커널[13]에 채택되었으며 다양한 IP 주소검색 연구의 기반이 되었다. 또 트라이 기반 연구들은 메모리 검색횟수를 감소시키기 위해 한 번에 한 비트가 아닌 다수의 비트를 비교한다. 즉, 트라이의 경로를 탐색할 때 복수개의 링크를 한번에 탐색하는 것처럼 동작한다. 그러나 이러한 탐색방법은 라우팅 정보를 갖지 않은 중간노드가 다수 포함되어 메모리의 사용효율이 저하되므로 메모리 사용효율을 높일 필요가 있다. Srinivasan 등은 [14]에서 IPv4에서 최대 32가지인 프리픽스 길이의 개수를 감축하도록 제한된 조건하에 프리픽스를 확장하고, 메모리 량을 최적화하기 위해 동적 프로그래밍 기법을 적용하여 트라이의 분할 개수와 분할 깊이를 결정하였다. [6]은 트라이 노드에 포함된 라우팅 프리픽스의 순서정보를 저장하는 비트 벡터를 트라이 레벨 16, 24 및 32에서 구성하여 메모리 크기가 캐시 크기보다 작게 함으로써 메모리 검색을 캐시 검색으로 변경하였다. 이때 IP 주소를 인덱스로 비트 벡터의 특정위치에 도착하고, MSB부터 특정위치까지의 비트 1의 누적 합은 Next-Hop의 인덱스이거나 다음 레벨의 비트 벡터에 대한 포인터 값이 된다.

[15]는 동일한 길이의 프리픽스로 그룹을 구성하고 각 그룹을 해시 테이블에 저장한다. 가장 긴 프리픽스의 그룹부터 해시 검색을 하고 차례로 짧은 길이의 프리픽스 그룹으로 검색을 진행한다. [16]은 프리픽스 검색에 이진검색을 적용할 수 있도록 프리픽스를 범위의 개념으로 전환하여 Low, High 프리픽스로 확장하였고, 캐시 워드 크기를 반영하여 multi-way 이진검색을 제안하였다. 프리픽스 분포가 균등하다고 가정할 때 검색속도는 Log_2N, multiway 검색속도는 log_mN이 된다. IP 주소검색에 사용된 다른 시도는 해시 테이블을 이용하는 것이다[17]. 해시 테이블은 프리픽스 전체를 저장하여야하고, 캐시에 저장되는 것을 전제로 하지만 백본 라우터의 해시 적중률은 별로 좋지 않다고 보고 되었다[18].

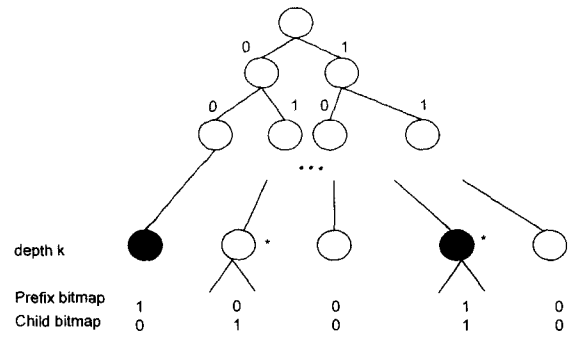
대량의 DRAM 메모리로 메모리 검색을 최소화하려는 연구로는 [8]에서 DRAM에 2-level 라우팅 테이블을 저장하는 방법을 들 수 있다. 24/8 방법으로 불리는 이 방법은 IP 주소의 MSB 24비트를 첫 번째 라우팅 테이블의 인덱스로 사용하고 나머지 8비트는 두 번째 라우팅 테이블의 오프셋(offset)으로 사용한다. 첫 번째 라우팅 테이블의 크기가 32 MB인 반면에 두 번째 라우팅 테이블의 크기는 작다. 최근의 동향[14]에서는 두 가지 목적을 동시에 달성할 수 있도록 자료구조의 크기와 메모리 검색횟수 사이의 균형 관계를 동적으로 탐색하고 그 결과를 반영하여 자료구조를 만드는 연구들이 발표되고 있다. 본 논문도 메모리의 크기와 검색속도의 상관관계를 파악하여, 시스템 캐시크기보다 작은 포워딩 테이블이 더 빠른 검색속도를 제공할 수 있으므로 비트맵 구조를 이용하여 메모리를 압축하고, [14]의 동적 레벨 선택기법과는 다르게 마지막 레벨에서만 동적 레벨을 사용하도록 변경하여 트라이 분할구조를 쉽고 안정적으로 구축할 수 있도록 하였다.

3. 동적 트라이 압축

동적 트라이 압축(DTC)은 트라이를 상하 2-단계로 분리하여 포워딩 테이블을 구성한다. 트라이 상위단계는 U-Trie로, 하위단계는 L-Trie로 구분되며 U-Trie는 프리픽스 정보와 하위의 L-trie로 연결되는 링크정보를 포함한다. L-Trie는 프리픽스 정보만 저장하며 U-Trie의 깊이보다 더 긴 프리픽스에 대해서만 생성된다. 그런데 (그림 3)에서 보듯이 IPv4의 실제 프리픽스 길이는 8~32 사이에서 분포되므로 L-Trie 엔트리가 생성되는 프리픽스의 길이는 다양하며 이때 프리픽스의 최대 길이 32를 기준으로 자료구조를 만들면 최대 길이에 미치지 못하는 대다수의 프리픽스에 대해서는 메모리를 낭비하게 된다. 메모리 낭비를 최소화하기 위해서 DTC는 L-Trie를 만들 때 능동적으로 L-Trie에 속하는 각 프리픽스의 최대길이를 측정하여 메모리의 낭비를 방지한다. 또한 U-Trie와 L-Trie를 분할하기위한 최적의 트라이 깊이는 필요한 메모리의 크기와 IP 주소검색 속도가 최소화되도록 능동적으로 조절되며, 트라이의 프리픽스와 링크정보를 표현할 때 메모리의 크기를 최소화하기 위해 각 정보를 하나의 비트로 표시하는 비트맵으로 압축한다.

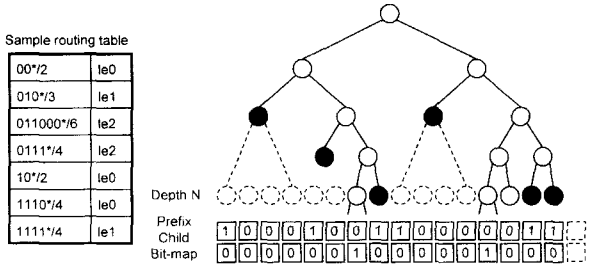
3.1 비트맵 자료구조

DTC는 트라이의 구조와 라우팅 정보 즉, 프리픽스 정보를 각각 하나의 비트로 표시하고 각 비트를 수집하여 비트맵을 만든다. (그림 1)에서 검은색 노드는 루트에서 출발하여 해당 노드에 도착하는 경로와 일치하는 라우팅 프리픽스의 정보를 갖는다. 이러한 이유로 검은색 노드를 프리픽스 노드라고 명명한다. 트라이의 프리픽스 노드의 존재유무를 0과 1로 표현하고 이 정보를 가진 비트를 수집하여 프리픽스 비트맵을 구성한다. 프리픽스 비트맵에서 각 비트 1은 프리픽스 노드까지의 경로와 일치하는 프리픽스가 존재함을 의미한다. 비트맵에서 비트 1이 프리픽스의 존재를 의미하므로 비트맵의 좌측으로부터 비트 1을 카운트한 값 즉, 비트 1의 누적개수는 트라이 전체에서 프리픽스의 순서를 의미하고 프리픽스의 순서는 Next-Hop 정보를 가진 해시 테이블의 인덱스가 된다.



(그림 1) 프리픽스 트라이와 Bitmap 구성 예제

두 번째 비트맵인 차일드 비트맵은 프리픽스 비트맵이 트라이 깊이 32에서 만들어진다면 필요 없다. 그러나 DTC는 2^{32}비트의 거대한 비트맵을 만들지 않고자하며, [14]에서 보인 2-단계 트라이 구조를 이용하는 2-단계 구조이므로 상위단계의 트라이보다 더 긴 프리픽스에 대해서는 L-Trie로 연결되는 정보가 필요하며 이 정보를 제공해주는 것이 바로 차일드 비트맵이다. (그림 1)의 예제에서는 프리픽스 비트맵과 차일드 비트맵의 구성방법을 보여주고 있으며, 두 번째와 네 번째 노드가 차일드 링크를 갖고 있으므로 차일드 비트맵에서 비트 값 1이 할당됨을 확인할 수 있다. 참고로 (그림 1)의 트라이는 완전이진 트라이[6]로 변경된 후의 모양이다.



(그림 2) 예제 라우팅 테이블과 프리픽스 및 차일드 비트맵 구성 예제

(그림 2)는 예제 라우팅 테이블에 따라 구성된 완전이진 트라이와 프리픽스 비트맵 및 차일드 비트맵 예제를 보여주고 있다. 트라이는 깊이 4까지만 표시되어 있고, 이것은 트라이 분리 단계가 4에서 형성된 경우로 해석할 수 있다. (그림 2)의 라우팅 테이블에서 IP 주소검색을 수행하는 과정을 간략히 설명하고자 한다. 어떤 패킷의 목적지 주소의 일부분이 '010000/6'이라고 하면, (그림 2)의 예제는 깊이 4에서 트라이가 분할되었으므로 주소의 좌측 4비트 '0100'을 검색키로 사용한다. '0100'은 인덱스 4로 해석하여 위 차일드 비트맵의 4번 원소를 검사한다. 이때 4번째 원소의 값이 0이므로 주어진 IP 주소에 대해 4비트 이상의 프리픽스는 존재하지 않는다는 것을 확인할 수 있다. 4비트 보다 더 긴 프리픽스는 존재하지 않으므로 프리픽스 비트맵을 검사하여 4비트 길이의 프리픽스가 있는지 여부를 검색한다. 차일드 비트맵과 마찬가지로 '0100'의 값을 인덱스로 사용하여 프리픽스 비트맵의 0번째 원소부터 4번째 원소까지 사이에 존재하는 비트 1의 개수를 카운트한다. 총 2개의 비트 1을 찾을 수 있으므로 2가 라우팅 테이블의 인덱스이다. 라우팅 테이블의 2째 프리픽스가 '010*/3'이 검색된 최장 프리픽스 엔트리이며, 동시에 le1이 결정된 다음-홉 주소가 된다. 만일 검색할 IP주소가 '011000'이라면 인덱스 6의 차일드 비트맵에서 비트값 1을 찾을 수 있다. 이것은 '011000' IP에 대해 4비트 이상의 프리픽스가 존재한다는 의미이므로 하위 트라이에 대해 검색을 계속하여야 한다.

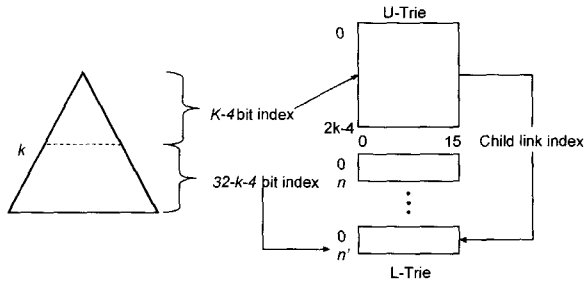
3.2 동적 트라이 압축

3.2.1 2-단계 트라이 구조

트라이는 해시와 함께 IP 주소검색을 위해 많이 사용되는 자료구조이다. 라우팅 테이블의 IPv4 프리픽스 트라이는 최대 깊이가 32로 라우터에 도착한 IP 패킷의 목적지 주소와 일치하는 프리픽스 노드를 찾기 위해 한 비트씩 비교 검색할 경우 최대 32회의 메모리 검색이 발생하여 속도가 매우 느려지므로 트라이 레벨을 압축[5]하거나 한번에 다수의 비트를 검색하는 방법이 연구되었다. 그러나 이러한 방법들의 문제점은 낮은 메모리 효율이다. 즉, 다수의 비트 k 비트를 한번에 비교 검색하기 위해서는 k 비트에 해당하는 트라이의 모든 정보를 저장하여야한다. 만일 k를 8로 가정하면 하나의 IP 주소는 8비트씩 4번의 검색을 하여야하며, 트라이의 8 비트에 해당하는 모든 정보를 저장하여야한다. 즉, 먼저 루트로부터 8비트인 트라이 깊이 8에서 256개의 노드정보를 모두 저장하고, 256개의 노드 각각에서 깊이 8을 더한 위치 즉, 깊이 16에서 256개의 노드정보를 저장하여야하며 깊이 24와 32에서도 마찬가지이다. 다만 깊이 8~24에서 노드정보를 저장할 때는 하위 노드로 연결되는 링크가 있을 때만 저장한다.

DTC는 2-단계 트라이 구조로 두 번의 검색으로 IP 주소 32비트를 검색한다. 즉, 상위 트라이 구조 U-Trie는 트라이 깊이 k에서 만들어진 비트맵이며, 하위 트라이 구조 L-Trie는 $32-k$ 비트의 정보를 저장한다. (그림 3)은 깊이 k에서 분할된 2-단계 트라이의 전형을 보여주고 있다. U-Trie는 깊이 k에서 만들어진 비트맵을 16비트씩 분할하여 엔트리를 구성한 배열구조의 해시 테이블이므로 크기는 2^{k-4}이다. (그림 3)에 표시된 것과 같이 IP 주소의 MSB $k-4$비트가 U-Trie 테이블의 인덱스가 되고 나머지 4비트가 비트별 위치정보가 된다. L-Trie는 $32-k$비트에 대한 정보를 갖는데, U-Trie의 노드가 깊이 k 아래로 차일드 노드를 가질 때만 만들어지며 U-Trie와 동일하게 16비트씩 분할된 비트맵을 하나의 엔트리로 갖는 테이블 구조이므로 $32-k$비트 중에서 $32-k-4$비트는 L-Trie에 대한 인덱스로, 나머지 4비트는 비

트별 위치정보가 된다. L-Trie는 U-Trie와 자식-부모의 관계가 있으므로 L-Trie를 검색할 때는 이미 검색한 U-Trie의 비트의 차일드 링크에 따라 만들어진 L-Trie를 검색하여야한다. (그림 3)에서 child link index로 표시된 링크를 따라 L-Trie 테이블을 선택한 뒤 *32-k-4*비트의 L-Trie 인덱스가 사용되는 예를 볼 수 있다.

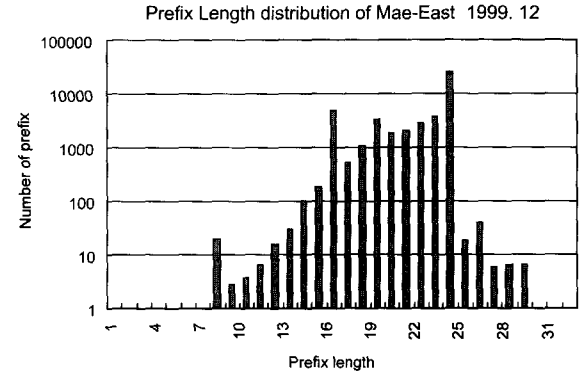


(그림 3) 2-단계 트라이 구조와 DTC 자료구조

2-단계 트라이 구조를 사용함으로써 얻을 수 있는 장점은 메모리 검색횟수의 감소를 꼽을 수 있다. (그림 3)처럼 두 번의 메모리 검색으로 32비트 주소를 모두 검색할 수 있다. 그러나 2-단계 구조의 가장 큰 단점은 자료구조에 필요한 메모리의 크기가 크다는 것이다. [14]에서는 2-단계 구조는 9.8MB가 필요하고, 6-단계 구조의 경우 420KB가 사용되었다. DTC의 경우 비트맵을 사용하므로 메모리의 크기를 [14]보다 많이 줄일 수 있지만 k의 선택에 따라 L Trie의 크기가 크다는 단점이 있으므로 메모리 크기를 줄이기 위한 방법이 필요하다. 참고로 자료구조의 크기가 L2 캐시보다 작아지면 메모리 검색이 캐시 검색으로 대체되는 효과를 얻을 수 있다. 참고로 U-Trie의 구조는 하나의 엔트리가 메모리에서 검색되면 L2 캐시워드에 엔트리가 완전히 업-로드될 수 있도록 U-Trie 엔트리를 설계한다. 이렇게 함으로써 엔트리에 대한 나머지 검색은 모두 캐시 검색이 된다.

3.2.2 트라이 분할 깊이 k와 메모리 크기의 관계

본 절에서는 2-단계 트라이 구조에서 단계를 구분하는 깊이 k의 선택에 따라 변화하는 메모리양을 살펴본다. DTC의 U-trie는 배열구조이므로 메모리의 크기는 깊이 k의 선택에 따라 지수적으로 증가하며 L-Trie의 크기는 $2^{32-k} \times$ nChild이며 이때 nChild는 깊이 k에 존재하는 차일드 링크의 개수이다. 다음의 <표 1>에서 k의 변화에 따른 L-Trie의 두 가지 크기를 볼 수 있다. 단순하게 계산한 L-Trie의 크기는 수십 MB로 매우 큰 값으로 k가 증가함에 따라 $32-k$가 감소하여 줄어들지만 트라이를 압축하여 얻을 수 있는 크기에 비해 여전히 큰 값이다. L-Trie의 크기를 압축하는 방법은 다음 절에서 설명한다. (그림 4)는 Mae-East 라우팅 테이블의 프리픽스 길이별 분포를 보여주는데 대부분의 프리픽스 길이가 8~24이며 25 이상의 길이는 매우 드문 경우임을 알 수 있다. 이것은 k의 값이 증가함에 따라 nChild도 증가함을 의미한다. 따라서 k가 증가하면 U-Trie와 L-Trie의 크기는 모두 증가한다. 그러나 k를 작은 값으로 선택하면 U-Trie에 포함되는 프리픽스 노드의 개수가 감소하여 L-Trie를 검색할 확률이 증가하므로 결과적으로 메모리 검색횟수가 증가하여 IP 주소검색 성능이 저하된다. 반대로 k를 큰 값으로 선택하면 자료구조의 크기는 커지지만 U-Trie에 포함되는 프리픽스 노드의 개수가 증가하여 IP 주소검색이 U-Trie에서 종료될 확률이 증가하여 성능이 좋아질 수 있다.



(그림 4) 프리픽스 길이 분포 예제

일반적인 시스템의 메모리 계층구조를 고려하면 k의 값을 무작정 큰 값으로 택함으로써 IP 주소검색 성능을 개선하려는 시도는 실패할 수밖에 없다. 즉, 포워딩 테이블 자료구조의 크기가 너무 커지면 자료구조에 대한 검색이 저속의 DRAM 메모리 검색이 되어 성능개선 효과가 저하된다. [14]의 연구와 본 연구의 실험결과는 자료구조의 크기가 L2 캐시보다 작을 때 더 좋은 성능을 얻을 수 있음을 보여준다. 결과적으로 k의 선택은 U-Trie에 보다 많은 프리픽스 노드가 포함되어 메모리 검색횟수를 최소화하여야 하며 동시에 자료구조의 크기가 L2 캐시보다 커지지 않도록 함으로써 저속의 DRAM 메모리 검색 대신 고속의 SRAM 캐시 검색이 많이 발생하도록 하여야 한다.

<표 1> Mae-East 테이블에서 k의 변화에 따른 L-trie 크기의 변화와 U-Trie의 프리픽스 개수

k	No. of Prefix in the U-Trie(KB)	nChild	Naive L-Trie Size(MB)	L-Trie Size(KB)
16	5736	3610	59.1	216.2
17	6979	5697	46.6	154.1
18	9382	8453	34.6	110.1
19	15239	10080	20.6	74.7
20	21157	12870	13.1	54.1

DTC는 적절한 k의 선택으로 최적의 메모리 크기와 검색

속도를 얻기 위해 동적으로 트라이를 탐색하여 k에 따른 메모리의 크기가 L2 캐시보다 작고, U-Trie에 포함된 프리픽스 노드의 개수가 최대가 되는 k를 탐색한다. 먼저 그림 4의 프리픽스 길이 분포에서 보면 16이하의 프리픽스 개수는 의미를 부여할 정도의 숫자가 되지 않으므로 k를 16부터 탐색하도록 하였다. k가 20일 때는 비트맵 하나의 크기가 2^{20}비트로 너무 커지므로 k의 상한선은 20으로 하였다. k가 21 이상 특히 24일 경우에는 거의 모든 주소검색이 U-Trie에서 종료되며 L-trie의 의미가 거의 없다고 볼 수 있으며, 특히 비트맵의 크기가 2^{21}비트 이상으로 커져서 L2 캐시보다 작은 메모리를 얻으려는 시도를 무의미하게 한다. k는 다음의 과정을 거쳐서 결정된다.

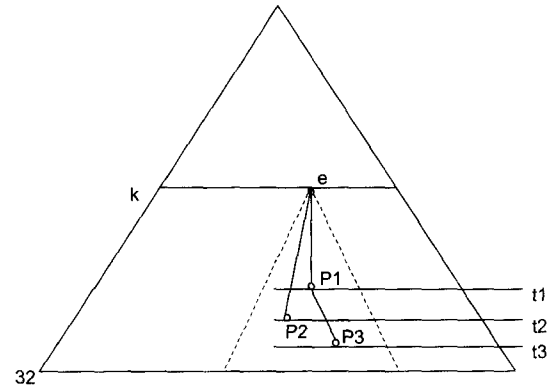
① 트라이를 깊이 16에서 넓이우선 탐색 순서로 각 노드를 검색한다.
② 깊이 16의 프리픽스 비트맵을 만든다.
③ 차일드 링크를 가진 노드는 그 노드를 루트로 깊이우선탐색을 한다.
④ 탐색의 결과 최대 깊이 *maxd*가 결정되고, 이 노드의 L-Trie 크기는 2^{maxd}가 된다.
⑤ U-Trie의 2^{16}과 L-Trie의 크기를 합하여 자료구조 크기를 결정한다.
⑥ ①~⑤의 과정을 깊이 17~20까지 반복한다. 단, 자료구조의 크기가 L2 캐시보다 크면 중단한다.

k를 계산하기 위한 과정이 끝나면 자료구조의 크기가 L2 캐시보다 작고 프리픽스 노드의 개수는 최대인 깊이를 k로 결정할 수 있다. 결정된 k는 앞에서 기술한 효과 즉, U-Trie에서 검색이 종료될 확률을 최대화하고 고속의 SRAM 검색효과를 충분히 얻을 수 있는 값이 된다. k의 결정 과정에서 4번째 스텝은 L-Trie의 크기와 매우 밀접한 관련이 있는 요소인 *maxd* 즉, 한 L-Trie의 최대 깊이를 탐색한다. (그림 4)에서 보듯이 거의 모든 프리픽스의 길이는 24이하인데 이러한 점을 감안하지 않고 L-Trie의 깊이를 32로 결정한다면 <표 1>의 단순히 계산한 L-Trie의 크기처럼 막대한 메모리를 사용하여야 한다. 물론 이 경우 IP 주소검색 과정에서 L-Trie를 검색하는 IP 주소의 길이를 $32-k$로 단순하게 결정할 수 있는 장점이 있지만 메모리 낭비를 보충하기에는 불충분한 것으로 판단된다. 위의 과정에서 각 k와 *maxd*의 값에 따라 메모리의 크기를 결정하는 식은 다음과 같다.

$$Memory\ size = c2^{k} + nChild \cdot c2^{maxd-k},$$
$$c = 2bit,\ and\ maxd = maximum\ depth\ of\ subtrie$$

3.2.3 동적 L-Trie 깊이 선택

*maxd*는 U-Trie와 L-Trie를 구분하는 깊이 k에 존재하는 노드 e의 차일드 프리픽스 노드 중에서 최대 깊이를 가진 노드의 깊이이다. (그림 5)에서 노드 e가 세 개의 프리픽스 노드를 포함하고 있으며 p3 노드가 최대 깊이이므로 t3가 노드 e의 *maxd*가 된다. *maxd*는 노드 e를 루트로 만들어지는 L-Trie의 비트맵의 범위를 한정하는 역할을 한다. t3가 노드 e의 차일드 노드 중에서 최대 깊이라는 것은 t3+1부터 32 사이에는 아무런 프리픽스 정보도 존재하지 않음을 의미하므로 L-Trie의 비트맵을 구성할 때 t3+1~32 사이의 비트맵을 구성할 필요가 없다는 것을 의미한다. 이것은 노드 e의 L-Trie 비트맵의 크기가 2^{32-k}가 아니라 2^{t3-k}로 변경됨을 의미한다. 예를 들어 k가 16일때 t3가 24라면 2^{16} 비트맵이 2^{8} 비트맵으로 축소되어 매우 높은 메모리 절감효과를 얻게 된다.



(그림 5) 노드 e의 L-Trie의 프리픽스 길이 분포

일반적인 라우팅 테이블의 프리픽스 길이 분포를 고려할 때 대부분의 *maxd*가 32에서 24이하로 변경됨에 따라 얻게 되는 메모리 절감효과는 매우 크다. <표 1>에서 *maxd*의 사용으로 줄어든 L-Trie의 메모리 크기를 확인할 수 있다. *maxd*를 사용하지 않은 경우와 비교하여 약 0.3% 크기로 감소하였다. 그러나 L-Trie를 검색할 때는 $32-k$비트를 사용하는 대신 $t3-k$비트가 사용되어야한다. 이것은 비트맵의 크기가 2^{t3-k}이기 때문이며, 깊이 k에 존재하는 모든 차일드 링크를 가진 노드들이 $k+1$~32의 값을 서로 다르게 가지므로 U-Trie에 차일드 비트맵을 저장할 때 함께 저장하여야 하며 U-Trie의 크기를 약간 늘어나게 한다. *maxd*는 3.2.2절에서 기술된 k 선택과정에 나타난 바와 같이 동적으로 각각의 차일드 노드마다 결정된다.

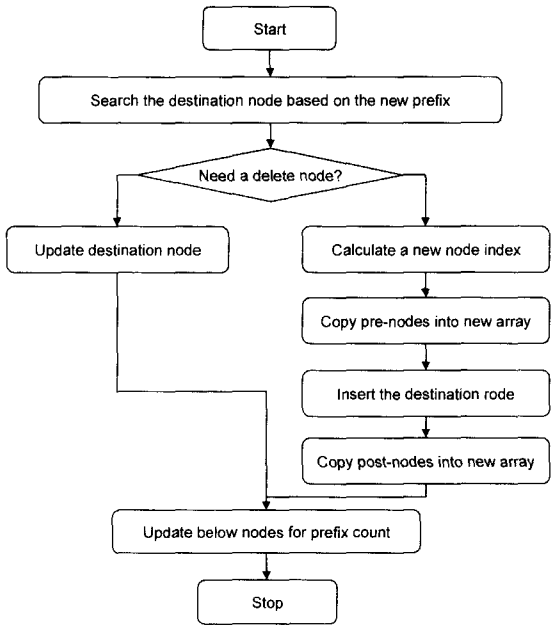
3.2.3 프리픽스의 삽입과 삭제

라우팅 테이블은 BGP와 같은 라우팅 알고리즘에 의해 계속 수정된다. 라우팅 테이블의 수정은 포워딩 테이블의 점진적(Incremental)인 수정을 요구하게 되며, DTC는 비교적 간단한 포워딩 테이블 수정 알고리즘을 지원한다. DTC 테이블 수정은 두 가지 경우, 새 프리픽스를 삽입하는 경우와 기존 프리픽스를 삭제하는 경우로 분류할 수 있다. 새 프리픽스 삽입 및 제거에 필요한 시간은 프리픽스 주소가 속한

DTC 테이블의 인덱스를 계산하는 과정과 새 노드의 삽입 또는 제거가 필요한지 결정하는 과정 및 DTC 테이블의 검색 및 수정에 소요되는 시간으로 구성된다. 세 가지 구성요소 중에서 가장 많은 시간을 차지하는 요소는 테이블의 검색 및 수정시간이므로 DTC 수정에 필요한 시간은 메모리 검색시간을 중심으로 기술하였다.

먼저 새 프리픽스를 삽입하는 경우는 기존의 DTC 테이블인 U-Trie와 L-Trie의 엔트리에 단지 하나의 프리픽스 정보를 추가로 표시하는 경우와 L-Trie를 새로 만들거나 수정하는 경우의 세 가지 경우가 있다. 첫째, U-Trie에 프리픽스 정보를 추가로 표시하는 경우에는 2^{k-4}개의 엔트리를 검색·수정하므로 수정되는 엔트리의 평균개수는 2^{k-6}개 이고, 이때 소요되는 시간은 DRAM 메모리의 평균 검색시간을 50ns로 가정하고 k를 16~20으로 하면 51.2~819.2μs이다. 여기에 약간의 계산시간이 추가된다. 둘째, L-Trie가 수정·생성되는 경우에는 2^{32-k}개의 엔트리가 검색되는데, 그림 4의 프리픽스 분포를 참고하면 이것은 최악의 경우이고 99% 이상의 경우 2^{24-k}개의 엔트리가 검색된다. 이때 필요한 시간은 k를 16~20으로 할 때 최악의 경우 3.2~0.2ms, 평균의 경우 12.8~0.8μs이다. U-Trie와 L-trie를 모두 수정해야할 경우에 필요한 시간은 U-Trie와 L-Trie 수정시간을 합해야하므로 최악의 경우 4ms, 평균의 경우 832μs이다.

단지 프리픽스 정보를 추가할 때에는 새 프리픽스 주소를 표시하는 비트가 표시될 비트-맵 테이블 노드 인덱스를 찾으면 되고, 새 테이블 노드를 삽입해야 하는 경우에는 새 노드를 삽입할 위치를 계산한 후에 배열로 구성된 기존 테이블을 다른 배열로 복사하는 과정이 필요하다. 이 과정을 정리하면 (그림 6)과 같이 표시할 수 있다.



(그림 6) DTC 포워딩 테이블의 수정

기존 프리픽스 정보의 삭제는 U-Trie의 프리픽스 정보를 단순히 삭제하고 관련정보를 수정하는 경우와, U-Trie의 수정이 L-trie의 수정을 동반하는 경우로 분리할 수 있다. U-Trie 엔트리만 수정될 경우에는 삽입과정과 마찬가지 경우가 되므로 수정되는 엔트리의 평균개수는 2^{k-6}개이고, 필요한 시간은 51.2~819.2μs이다. L-trie의 수정은 두 가지 경우로 나누어지는데 L-Trie가 수정되는 경우와 L-Trie가 삭제되는 경우가 있다. L-Trie의 수정은 삽입과 마찬가지이므로 최악의 경우 3.2~0.2ms, 평균의 경우 12.8~0.8μs의 시간이 필요하다. L-Trie 삭제의 경우는 단순히 인덱스를 제거하면 되므로 소요시간을 계산할 필요가 없다고 판단된다.

4. 실험 결과

DTC 실험은 3장에서 기술한 k의 변화에 따른 자료구조의 크기변화를 검사하고, 자료구조의 크기와 L2 캐시 크기보다 작은 크기의 자료구조 크기가 IP 주소검색 성능에 어떠한 영향을 미치는가를 실험하였다. 본 실험은 지금은 종료된 IPMA[19] 프로젝트의 백본 라우팅 테이블을 대상으로 하였다. 실험에 투입된 IP 주소는 랜덤으로 만들어진 주소이며, 한번의 실험에 백만 개의 주소를 투입하였고 얻어진 IP 주소 검색 시간은 총 수행시간을 투입된 IP 주소의 개수로 나눈 평균값이다. 실험에 사용된 플랫폼은 펜티엄 II 450MHz CPU에 L2 캐시 크기는 512KB이다.

4.1 메모리 크기와 검색속도

<표 2>에 실험에 사용된 IPMA 라우팅 테이블에 대한 정보를 정리하였다. 자료는 1999년도에 수집된 것으로 Aads와 Paix를 제외한 나머지 테이블들은 대형 백본 라우팅 테이블의 크기를 갖고 있다. 원래의 프리픽스 개수와 DTC에 저장된 프리픽스의 개수가 틀린 것은 라우팅 테이블을 트라이로 변경할 때 완전이진 트라이[6]로 변경되기 때문으로 새 프리픽스가 생성된 것은 아니고 중복 프리픽스가 생성된 것이다. <표 3>에는 IPMA 라우팅 테이블별로 DTC 자료구조의 크기와 자료구조 생성에 소요된 시간을 정리하였다. DTC 생성시간은 트라이가 만들어진 후 k를 결정하고 DTC 비트맵을 만드는 시간이다. 생성시간은 테이블별로 30ms~120ms가 기록되었는데 테이블의 크기가 작을수록 적은시간이 사용되며, 같은 크기에서는 L-Trie의 개수가 많을수록

〈표 2〉 IPMA 라우팅 테이블(1999. 12)의 프리픽스 개수

	No. of Original Prefix	No. of Prefix in the DTC
Mae-East	48290	89621
Mae-West	32462	56329
PacBell	26897	44136
Paix	18093	16768
Aads	10434	28361

〈표 3〉 IPMA 라우팅 테이블과 DTC 자료구조의 크기 : Size(KB), Average Build Time(ms)

k	Mae-East			Mae-West			PacBell			Paix			Aads		
	DTC Size	Build Time	No. of L-Trie	DTC Size	Build Time	No. of L-Trie	DTC Size	Build Time	No. of L-Trie	DTC Size	Build Time	No. of L-Trie	DTC Size	Build Time	No. of L-Trie
16	371	100	3610	320	70	3118	270	70	2883	178	30	1784	224	50	2415
17	374	110	5697	322	80	4808	285	80	4346	208	40	2428	246	60	3444
18	461	100	8453	409	80	6741	382	70	5960	318	30	3004	347	50	4480
19	688	110	10080	639	80	7630	618	80	6713	565	40	3143	588	50	4895
20	1192	120	12870	1146	110	9313	1127	90	8163	1081	50	3547	1101	70	5768

L-Trie 생성에 많은 시간이 소모되므로 전체 생성시간은 커진다.

<표 4>는 라우팅 테이블별로 k의 변화에 따르는 검색성능을 보여준다. 기록된 검색시간은 하나의 IP 주소를 검색하는 시간으로 백만 개의 랜덤 IP 주소를 실험하여 얻은 평균값이다. 검색성능은 테이블의 크기와는 무관하며 자료구조의 크기와 밀접한 관련이 있다는 것을 알 수 있다. 다시 <표 3>을 참조하면 k가 16~18의 범위일 때 자료구조의 크기는 펜티엄II 프로세서의 L2 캐시 크기보다 작고 19, 20의 경우에는 더 큰데 이때 성능의 차이가 있음을 확인할 수 있다. 그런데 k가 17과 18일 때는 모두 L2 캐시보다 작은 크기이지만 성능의 차이가 크게 나타났다. 이것은 k가 18일 때의 자료구조 크기가 L2 캐시 보다는 작지만 실제로는 이 자료구조가 저장되는데 실제 사용된 캐시의 크기보다 커져서 캐시 미스가 많이 발생했기 때문으로 판단된다. (그림 7)의 그래프에서 k가 17, 18일 때 이러한 판단이 적절한 것임을 확인할 수 있다.

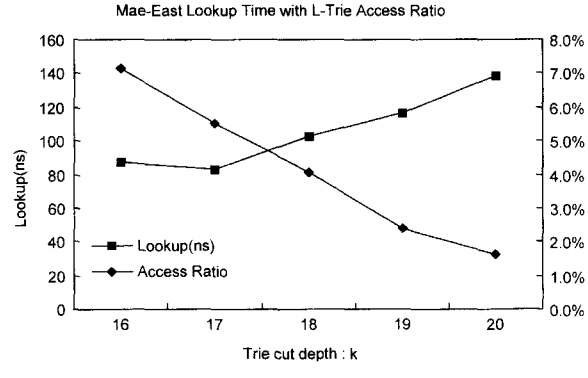
〈표 4〉 IPMA 라우팅 테이블의 평균 검색성능

(단위 : ns)

k	Mae-East Lookup Time	Mae-West Lookup Time	PacBell Lookup Time	Paix Lookup Time	Aads Lookup Time
16	87	90	80	90	90
17	83	80	80	80	80
18	103	100	110	100	90
19	116	110	110	110	110
20	139	140	140	130	140

(그림 8)에서는 검색성능과 L-Trie에서 검색이 완료된 비율을 보여주고 있는데, 참고로 IPMA 라우팅 테이블은 디폴트(default) 엔트리 라우팅이 없고 가장 많은 비트가 일치하는 엔트리를 선택한다는 것이다. (그림 8)에서 k가 증가할수록 L-Trie에서 검색이 완료되는 비율이 감소함을 확인할 수 있는데 이것은 k가 증가할수록 U-Trie에 더 많은 프리픽스 정보가 포함되기 때문이다. 따라서 L-Trie에서 검색이 종료되는 비율이 감소할수록 메모리 검색횟수가 감소하므로 검색성능은 좋아져야한다. 그러나 그래프에서는 대체로 반대 현상을 보여주고 있다. 이것은 k가 증가할수록 자료구조의 크기가 커져서 L2 캐시 검색의 효과가 떨어지기 때문이다. k가 16과 17일 때는 자료구조의 크기가 모두 충분히 작으므로 k = 17에서 L-Trie 검색비율이 감소함에 따라 검색시간이 단축되었다.

(그림 7) IP 주소검색 성능과 자료구조의 크기

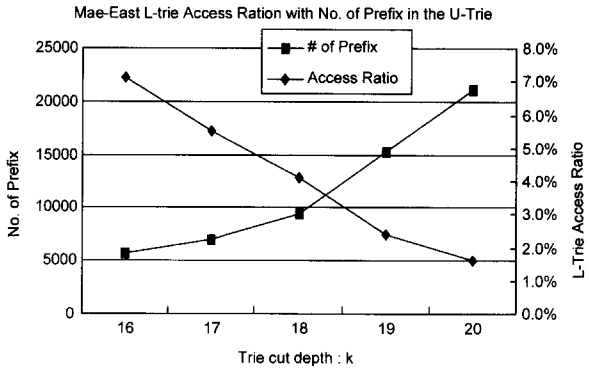


(그림 8) IP 주소검색 성능과 L-Trie 검색비율

DTC IP 주소검색은 검색대상 프리픽스의 길이가 k보다 짧아서 U-Trie에서 검색이 종료될 경우 U-Trie의 프리픽스 비트맵과 U-Trie Next-Hop 테이블에 대한 두 번의 메모리 검색과 차일드 링크가 있는지 판단하기 위해서 U-Trie의 차일드 비트맵을 한번 검색한다. 이때 차일드 비트맵 검색은 캐시 검색이므로 110ns/IP 성능을 계산할 수 있다. 참고로 메모리 읽기속도는 50ns, 캐시 읽기속도는 10ns로 가정한다. 이때 메모리 검색이외의 계산에 필요한 시간이 추가되어야 하므로 실제 검색속도는 조금 더 느려질 것이다. IP 주소

검색이 L-Trie에서 종료된다면 메모리 검색횟수는 U-Trie의 차일드 비트맵, L-Trie 및 L-Trie Next-Hop 테이블 검색의 3번이 된다. 또한 L-Trie 검색에 필요한 비트길이를 결정하기 위해 L-Trie의 최대깊이를 U-Trie에서 찾기 위해 한번의 캐시 검색이 발생한다. 결과적으로 이때의 검색시간은 160ns/IP이며 여기에 부가적인 계산시간이 추가된다. 이러한 계산결과를 <표 4>에 기록된 값과 비교하면 DTC 자료구조의 크기를 압축하여 L2 캐시보다 작게 함으로써 얻는 캐시 검색의 효과가 충분히 크다는 것을 알 수 있다.

동적으로 k를 검색하고 선택함으로써 DTC 자료구조의 크기를 조절할 수 있고, 자료구조의 크기가 L2 캐시보다 충분히 작을 경우 IP 주소검색의 성능이 대폭 개선되는 것과 k를 증가시켜 U-Trie에 포함된 프리픽스의 개수를 증가시킴으로써 주소검색 성능이 개선되는 것을 확인하였으므로 우리는 k의 결정기준을 명확하게 할 수 있다. (그림 9)의 그래프에서 나타난바와 같이 U-trie에 포함된 프리픽스의 개수가 증가할수록 검색시에 L-trie에서 검색이 종료되는 비율은 감소하므로 k는 자료구조의 크기가 L2 캐시보다 작고 프리픽스의 개수가 많은 값을 선택한다. 그러나 그림 8에서와 같이 메모리의 증가는 L-trie 검색횟수의 증가보다 검색성능에 훨씬 더 나쁜 영향을 미치므로 자료구조의 크기에 더 큰 가중치를 주어서 한 단계 더 작은 크기의 k 값을 선택한다. 본 실험의 경우 이러한 원칙을 적용하면 모든 라우팅 테이블에서 k는 17이 되고 이때 가장 좋은 성능을 기록하였다.



(그림 9) U-trie의 프리픽스 노드 개수와 L-trie 검색 비율

4.2 기존 검색 알고리즘과의 비교

<표 5>는 기존의 몇몇 연구들과 DTC의 성능을 비교한 것이다. 기존 연구와 성능을 비교할 때는 각각의 연구들의 특징을 살펴볼 필요가 있다. [16]과 테이블을 자료구조의 크기를 압축하여 성능을 개선하는 점이 비슷한 점이고, 다른 연구들은 메모리 검색횟수를 줄이는데 치중하고 있다. 이러한 두 가지 종류의 연구를 <표 5>에서 비교하면 [5]는 3.2MB의 포워딩 테이블을 해시 테이블로 구성하여 비교적 빠른 속도를 얻고 있다. 하지만 [14]는 완전 해시를 위한 해시함수 구성에 필요한 시간이 13분 정도로 매우 길다는 약점이 있다. 참고로 DTC를 제외한 나머지 연구의 성능은 [5]에서 인용하거나 펜티엄 II 프로세서로 속도를 환산한 것이다. [14]의 연구는 프리픽스 길이를 최소화하도록 프리픽스를 정밀하게 확장하고 동일한 길이의 프리픽스를 하나의 노드로 트라이를 만든다. 또한 동적 프로그래밍 기법을 적용하여 트라이의 분할 개수와 깊이를 결정한다. 본 논문은 [14]에서 4-단계 트라이 분할이 가장 좋은 성능을 기록하였지만 비트맵을 사용할 경우 2-단계 구조가 메모리 검색횟수를 줄이는데 더 유리하므로 2-단계 구조를 사용하였다. 본 연구의 성능이 펜티엄 II 450에서 80~140ns이므로 CPU의 속도를 감안하더라도 많은 성능향상이 있었음을 알수 있다.

〈표 5〉 기존 검색 알고리즘의 성능

	Size(KB) and # of prefix	Average Lookup time(ns)/packet
[6]	160/32,732	652/PII 400MHz
[11]	800/41,578	1000/PII 300MHz
[14]	500/38,816	196/PII 300MHz
[15]	1,200/33,000	650/PII 300MHz
[16]	950/38,816	430/PII 300MHz
DTC	245/48,290	140/PII 450MHz

5. 결론 및 향후 연구방향

본 연구에서 개발한 DTC 자료구조 및 검색 알고리즘은 트라이를 기반으로 한 2-단계 구조의 비트맵 포워딩 테이블이다. DTC는 적절한 자료구조의 크기와 상위단계의 U-Trie에 프리픽스 정보가 더 많이 포함되도록 함으로써 고속의 IP 주소검색을 지원할 수 있으며, 트라이를 2-단계로 분할할 때 자료구조의 크기를 측정하여 고속의 SRAM 캐시 검색을 최대한 사용할 수 있는 적절한 분할 깊이 k를 동적으로 결정할 수 있었다. 또한 k의 변화에 따라 U-Trie와 L-Trie이 포함되는 프리픽스의 비율을 조절함으로써 IP 주소검색이 U-Trie에서 완료되는 비율과 검색성능의 관계를 밝힘으로써 비슷한 크기의 자료구조 크기에서 k를 선택하는 기준을 제시하였다. 실험결과 기존 IP 주소검색 알고리즘과 비교해서 작은 메모리 소요량과 더 빠른 검색을 할 수 있음을 보였다.

DTC 자료구조의 향후 연구는 IPv6의 128비트 주소에 대해 다단계 비트맵 구조를 적용할 수 있는지를 실험하고, 최적의 단계수와 단계의 분할 깊이를 동적으로 측정하는 것이 첫 번째 과제이다. 두 번째 연구는 L2 캐시보다 작은 자료구조의 크기가 캐시 검색효과를 얻고 있음을 수치적으로 증명하고 더 나은 캐시 활용방법을 도출하기 위해 주소를 검색할 때 발생하는 L2 캐시 적중률을 측정하고 분석하는 것이다. DTC에서 이미 U-Trie와 L-Trie의 엔트리 크기를 캐시 엔트리 크기이하로 조절하고 있지만 이러한 방법의 효율성을 분석하기 위해 캐시 적중률을 측정할 필요가 있다.

참 고 문 헌

[1] V. Fuller, T. Li, J. Yu and K. Varadhan, "Classless Inter Domain Routing(CIDR) : an Address Assignment and Aggregation Strategy," RFC 1519, IETF, Spt., 1993.

[2] Y. Rekhter and T. Li, "A Border Gateway Protocol 4 (BGP-4)," RFC 1771, IETF, Mar., 1996.

[3] A. J. McAuley and P. Francis, Fast routing table lookup using CAMs, Proc. IEEE Infocom '93, San Francisco, 1993.

[4] N. Huang and S. Zhao, "A Novel IP-Routing Lookup Scheme and Hardware Architecture for Multigigabit Switching Routers," IEEE JSAC, Vol.17, No.6, Jun., 1999.

[5] S. Nilsson and G. Karlsson, "Fast Address Look-up for Internet Routers," Proc. of IEEE Broadband Communications '98, Apr., 1998.

[6] M. Degermark, A. Brodnik, S. Carlsson and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," Proc. of ACM SIGCOMM '97, Oct., 1997.

[7] Y-C. Liu and C-T. Lea, "Fast IP Table Lookup and Memory Reduction," 2001 IEEE Workshop on High Performance Switching and Routing, 2001.

[8] P. Gupta, et al., "Routing Lookups in Hardware at Memory Access Speeds," Proc. of IEEE Infocom '98, San Francisco, Apr., 1998.

[9] E. Rosen, A. Viswanathan and R. Callon, "Multiprotocol Label Switching Architecture," RFC 3031, IETF, Jan., 2001.

[10] T. Cormen, C. Leiserson and R. Riverst, "Introduction to Algorithms," The MIT Press, 1990.

[11] S. Nilsson and G. Karlsson, "Fast Address Lookup for Internet Routers," Proc. of IEEE Broadband Communication '98, Apr., 1998.

[12] G. Gonnet and R. Baeza-Yates, "Handbook of Algorithms and Data Structures," 2nd Ed., Addison Wesley, 1991.

[13] K. Sklower, "A Tree-Based Routing Table for Berkeley Unix," Proc. of the Winter Usenix Conference, 1991.

[14] S. Venkatachary and G. Varghese, "Faster IP Lookups using Controlled Prefix Expansion," Proc. of ACM Sigmetrics, Sep., 1998.

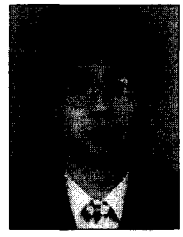
[15] M. Waldvogel, G. Varghese, J. Turner and B. Plattner, "Scalable High Speed IP Routing Lookups," Proc. of ACM SIGCOMM '97, Oct., 1997.

[16] B. Lampson, V. Srinivasan and G. Varghese, "IP Lookups using Multiway and Multicolumn Search," Proc. of INFOCOM, Mar., 1998.

[17] Torrent Networking Technologies Corporation, "High-Speed Routing table Search Algorithms," A technical paper, http://www.torrentnet.com.

[18] P. Newman, G. Minshall and L. Huston, "IP Switching and Gigabit Routers," IEEE Communications Magazine, Jan., 1997.

[19] Michigan University and merit Network, Internet Performance Management and Analysis (IPMA) project, http://nic.merit.edu/~ipma.



오 승 현

e-mail : shoh@donguk.ac.kr

1988년 동국대학교 전자계산학과(학사)

1998년 동국대학교 컴퓨터공학과(석사)

2001년 동국대학교 컴퓨터공학과(박사)

1987년~1996년 (주)대우엔지니어링

2001년 (주)세이프텍 연구소장

2002년~현재 동국대학교 컴퓨터학과 전임강사

관심분야 : 실시간 프로토콜, 홈 네트워킹, 센서 네트워크, IPv6 등