고속 라우터를 위한 향상된 비트맵 룩업 알고리즘

이 강 우[†] · 안 종 석[††]

요 약

인터넷 회선의 고속화에 따라 패킷을 초당 기가비트 이상의 속도로 전송하는 라우터에 대한 연구가 활발하다. 본 논문에서는 라우터의 병목인 최장 프리픽스 검색(LPM : longest prefix matching)을 효율적으로 수행하기 위해 제안된 비트맵 트라이(Bitmap Trie) 알고리즘을 혁신적으로 향상시킨 방법을 제시한다. 이 방법은 검색 시간을 단축하기 위하여 다음과 같은 세 가지 기법을 적용하였다. 첫째, 카운트 테이블을 추가하여 기존 알고리즘에서의 과도한 시프트 연산을 제거하여 프로세서 내의 처리시간을 단축했다. 두 번째는 트랜스퍼 테이블내의 중복된 포워딩 정보를 제거하여 메모리 효율을 증가시켰으며, 마지막으로, 포워딩 정보에 대한 접근빈도에 따라 검색범위를 다원화하여 데이터 접근을 최적화하였다. 또한, 신뢰도가 가장 높은 실행-구동 시뮬레이션 방법을 채택함으로써 검색의 지연요소를 발생 원인별로 세분하여 알고리즘이 실행되는 과정을 면밀하게 분석할 수 있었다. 실험 결과는 실제 시스템으로부터 실측한 결과와 비교하는 검증과정을 거쳤으며, 그 결과 기존 알고리즘에 비해 검색 수행시간을 약 82% 단축한 알고리즘을 개발하였다.

Enhanced Bitmap Lookup Algorithm for High-Speed Routers

Kangwoo Lee[†] · Jongsuk Ahn[††]

ABSTRACT

As the Internet gets faster, the demand for high-speed routers that are capable of forwarding more than giga bits of data per second keeps increasing. In the previous research, Bitmap Trie algorithm was developed to rapidly execute LPM(longest prefix matching) process which is well known as the severe performance bottleneck. In this paper, we introduce a novel algorithm that drastically enhanced the performance of Bitmap Trie algorithm by applying three techniques. First, a new table called the Count Table was devised. Owing to this table, we successfully eliminated shift operations that was the main cause of performance degradation in Bitmap Trie algorithm. Second, memory utilization was improved by removing redundant forwarding information from the Transfer Table. Lastly, the range of prefix lookup was diversified to optimize data accesses. On the other hand, the processing delays were classified into three categories according to their causes. They were, then, measured through the execution-driven simulation that provides the higher quality of the results than any other simulation techniques. We tried to assure the reliability of the experimental results by comparing with those that collected from the real system. Finally the Enhanced Bitmap Trie algorithm reduced 82% of time spent in previous algorithm.

키워드 : 고속 라우터(High-speed Router), 포워딩 테이블(Forwarding Table), 최장 프리픽스 검색(Longest Prefix Matching), 비트맵 트라이(Bitmap Trie), 실행-구동 시뮬레이션(Execution-Driven Simulation)

1. 서 론

인터넷 회선이 고속화됨에 따라 패킷을 초당 기가비트 이상의 속도로 처리하고 전송하는 라우터를 개발하기 위한 연구가 활발히 진행되고 있다. 라우터는 수많은 호스트들의 패킷이 경유하는 중간에 위치하며, 입력된 패킷을 헤더에 명시된 최종 목적지에 도착할 수 있도록 최적의 포트로 출력하는 역할을 수행한다. 이와 같이 출력포트를 결정하는 과정은 라우팅 테이블을 이용한 최장 길이 프리픽스 검색(LPM : Longest Prefix Matching)[13]을 통하여 이루어진다. 그런데, 이 과정에는 오랜 시간이 소요되어 네트워크의 병목이 되므로 라우터의 고속화에 대한 연구는 프리픽스 검색시간 단축에 초점이 맞추어지고 있다.

라우터의 프리픽스 검색속도 개선에 대한 연구는 하드웨어, 프로토콜 또는 소프트웨어에 관련된 연구로 구분된다. 하드웨어에 관련된 연구는 라우터의 구조에 관한 연구가 주로 이루어지고 있다. 검색을 실행하는 엔진을 각 라인카드로 분산시키거나, 다중 프로세서를 이용한 병렬 라우팅을 통하여 부하를 공정하게 분산시켜 라우터의 속도를 향상시키는 연구가 여기에 속한다. 빠른 접근시간을 갖는 메모리를 이용하는 방법에 대한 연구도 이 범주에 속한다. 프로토콜에 관련해서는 IPv4에 규정된 패킷의 헤더정보 대신 추

† 정 회 원 : 동국대학교 정보통신공학과 교수
†† 종신회원 : 동국대학교 컴퓨터공학과 교수
논문접수 : 2004년 2월 3일, 심사완료 : 2004년 2월 25일

가적인 헤더를 사용하거나 패킷을 재조립하는 등 검색을 회피하는 방법들이 연구되고 있다. 예를 들어 ATM(Asynchronous Transfer Mode), MPLS(Multiprotocol Label Switching), IP 스위칭 등이 있다. 끝으로 소프트웨어에 관한 연구는 라우팅 정보를 효율적으로 압축하여 저장할 수 있는 데이터 구조를 개발하는 연구가 있다. 그리고 검색과정에서 발생하는 지연시간을 최소화하여 검색을 신속하게 수행할 수 있는 알고리즘을 개발하는 연구도 여기에 포함된다.

본 논문에서는 소프트웨어적인 측면에서 라우터의 성능을 향상시키고자 하였다. 즉, 라우팅 테이블을 압축하는 방법 중 하나인 비트맵 트라이(Bitmap Trie)를 이용한 검색 알고리즘[12]이 안고 있는 문제점을 개선하여 성능을 혁신적으로 개선한 방안을 소개한다. 첫째, 카운트 테이블이라는 256바이트 용량의 소규모 테이블을 추가하여 기존 알고리즘에 있어서의 과도한 시프트 연산을 제거하였다. 둘째, 주소 검색과정에서 출력 포트정보를 저장하고 있는 트랜스퍼 테이블에서 중복된 라우팅 정보를 제거함으로써 메모리 효율을 향상시켰으며, 끝으로 주소 검색범위를 다원화하여 데이터 접근 횟수를 최적화하였다.

한편, 성능을 평가하는 일반적인 방법은 시뮬레이션을 이용하거나 실제 시스템을 구축하여 성능을 측정하는 것이다. 그러나 전자의 경우에는 시뮬레이션 결과에 대한 신뢰도가 낮고, 후자의 경우에는 내부에서 발생하는 현상파악이 어려워 병목의 위치와 원인을 진단하기 어렵다.

이에, 본 연구에서는, 시뮬레이션의 단점을 극복하기 위하여, 결과에 대한 신뢰도가 가장 높은 실행-구동 시뮬레이션(execution-driven simulation)을 수행하였으며 실험결과를 실측결과와 비교하는 검증과정을 거쳐 결과에 대한 신뢰성을 확보하였다. 또한, 이 시뮬레이션 방법을 통하면 프리픽스 검색과정에서 발생하는 내부현상을 정확하게 파악할 수 있다. 이러한 장점에 힘입어, 본 연구에서는 검색에 소요되는 지연요소를 그 발생원인에 따라 명령어 처리지연, 캐시 접근지연 및 메모리 접근지연으로 구분하였다. 이와 같은 면밀한 성능진단을 토대로 각 지연요소를 유발하는 원인을 최소화함으로써 주소 검색시간이 기존의 알고리즘보다 약 82% 단축된 알고리즘을 개발하였다.

본 논문의 제2장에서는 프리픽스 검색을 고속화하기 위한 관련연구들을 요약하고, 제3장에서는 기존의 비트맵 트라이를 이용한 알고리즘을 간단하게 소개한다. 제4장에서는 기존의 알고리즘에 대한 분석과 이의 성능을 향상시키기 위하여 본 연구에서 새롭게 제시하는 향상된 비트맵 트라이 알고리즘에 관하여 자세히 설명한다. 제5장에서는 실행-구동 시뮬레이션을 이용한 실험방법에 대하여 설명하고, 제6장에서는 각 알고리즘에 대한 실험결과의 비교, 분석을 통하여 본 연구에서 제안하는 알고리즘의 성능을 검증한다. 마지막으로 제7장에서는 결론과 향후 연구 과제를 제시한다.

2. 관련 연구

라우팅 속도 개선에 관한 연구는 하드웨어, 프로토콜 또는 소프트웨어 중심의 연구로 나뉘어 진다. 하드웨어 중심의 연구는, 다시 라우터 구조 연구와, 메모리에 관한 연구로 크게 나누어진다. 라우터 구조에 관한 연구는 주로 하나의 라우터에 다중 프로세서를 장착하여 라우터의 속도 향상을 도모한다. 또한 메모리에 관한 연구는 접근시간이 큰 DRAM을 대신 할 수 있는 고속 메모리와, 고속의 검색을 가능하게 하는 하드웨어에 관한 연구 등을 포함한다. 프로토콜에 관한 연구는 주로 주소 검색을 회피하는 기법을 표준화하는데 초점이 맞추어지고 있다. 그리고 소프트웨어에 관한 연구에는 포워딩 테이블의 데이터구조와 그에 따르는 검색 알고리즘에 대한 연구가 포함된다.

2.1 하드웨어 기반 연구

하드웨어에 관련된 연구는 우선 라우터의 구조에 관한 연구가 주로 이루어지고 있다. 우선, 대용량의 트래픽을 처리하기 위하여 주소 검색을 실행하는 포워딩 엔진을 각 라인카드로 분산시키는 구조가 도입되었다[17]. 이 경우 특정 라인카드의 포워딩 엔진에 부하가 집중되는 경우가 발생하기도 한다. 이러한 부하를 공정하게 분산시키고 라우터의 전체 시스템 능력을 향상시키고자 하는 다양한 방안이 소개되었다. 예를 들어 라인카드에 포워딩 엔진을 삽입하는 분산 라우터 구조에서는 분류작업과 출력 스케줄링을 라인카드에서 담당하고 주소 검색은 포워딩 엔진에서 담당한다. 현재 백본 망에서 많이 사용되어지는 분산 포워딩 라우터 구조는 멀티 프로세싱이 가능하기 때문에 라우터의 성능을 크게 향상시킬 수 있으나, 공정하지 못한 패킷의 입력으로 인하여 패킷의 지연시간이 균등하지 못하고, 많은 양의 시스템 자원을 사용하여야 하므로 비용의 증가를 가져온다[6]. 이러한 시스템의 성능 저하를 최소화하고 자원의 낭비를 막기 위한 대안으로 병렬 포워딩 라우터 구조가 제시되었다. 이 구조에서는 하나의 라인카드로 들어온 패킷이 부하가 적은 포워딩 엔진에서 처리되기 때문에 라인카드에서의 지연시간을 줄일 수 있으며, 불필요한 자원의 낭비를 막을 수 있다[2]. 이와 같은 라우터의 구조에 관한 연구에는 스위치 구조와 공유 메모리 구조 및 확장성에 관한 연구가 동시에 이루어진다.

한편, 10기가비트 라우터를 기준으로 할 때 평균 초당 1천만 피킷을 처리하기 위해서는, 100ns 안에 입력된 패킷의 출력 포트가 정해져야 한다. 그러나 포워딩 테이블을 DRAM에 저장할 경우, 메모리 접근시간이 60~100ns정도가 소요되므로 이러한 목표를 달성하기 불가능하다. 따라서 보다 빠른 처리시간을 갖는 메모리에 대한 연구가 필요한데, 이 분야의 연구의 대부분은 CAM(Content-Addressable Memory)이나 고속의 SRAM을 이용한다. CAM이나 SRAM은 고가이지만 전자는 접근방법 측면에서 후자는 접근시간 측면에서 빠른 처리속도를 갖는다는 장점이 있다.

McAuley *et al.*[1, 10]은 프리픽스를 길이별로 각각의 CAM에 저장한 후 들어온 패킷의 도착지 주소와 병렬로 비교 검색한다. 따라서 32비트 주소를 검색하기 위해서는 32개의 CAM이 필요하며 앞으로 일반화될 128비트 IP주소의 경우 최대 128개의 CAM이 필요하다. 따라서 CAM을 장착한 시스템은 매우 고가의 장비가 된다. 또한 CAM을 사용하는 경우에는 프리픽스를 길이에 따라 CAM에 분리하여 저장해야 하고, 라우팅 정보의 갱신동작이 복잡하다는 단점이 있다.

SRAM은 접근시간이 빠르지만 고가이므로 대용량의 포워딩 정보를 요구하는 백본용 라우터에는 적합하지 않다. 다만 포워딩 테이블의 크기가 SRAM에 적합할 정도로 작아서 SRAM을 이용한 주소 검색을 할 수 있다면 검색 속도의 향상을 가져올 수 있을 것이다. Hwang *et al.*[2]에서는 SRAM을 이용하기 위해 포워딩 테이블을 SRAM 크기에 맞추는 연구를 하였다. 그 결과 약 40,000개의 라우팅 엔트리를 450~470KBytes에 저장할 수 있었다. 그러나 주소 검색을 실행하기 위한 알고리즘이 복잡하며 프리픽스의 길이가 24이상인 경우 포워딩 테이블의 크기가 1.5MBytes를 초과하여 대용량의 SRAM을 필요로 한다.

또한, 캐시를 이용하는 방법도 있다. 그러나 캐시란 데이터의 지역성(locality)이 우수한 경우에만 유용하므로, 인터넷 초기에는 이 캐시를 이용한 방법도 가능했겠으나, 급속도로 팽창한 인터넷 환경에서는 지역성이 매우 낮게 관찰된다. 이러한 경우 캐시의 접근 성공률을 높이기 위해서는 대용량의 캐시가 요구되므로 비경제적이다. Chvets *et al.*[3]에서는 캐시의 공간지역성을 이용해 주소 검색의 실패율을 낮추고자 하였다. 즉 캐시를 2개 지역으로 나누어 각각 다른 프리픽스 길이별로 저장하였으나 하나의 지역으로만 사용했을 때보다 실패율이 크게 향상되지 못했다.

2.2 프로토콜 기반 연구

프로토콜 기반 연구는, 각 라우터에서 포워딩을 하는데 있어서 IPv4에 규정된 패킷의 헤더정보를 사용하지 않고, 추가적인 헤더를 사용하거나, 패킷을 재조립하여, IP주소 검색 자체를 회피하는데 그 목적이 있다. 이 분야의 연구에는 ATM Asynchronous Transfer Mode), MPLS(Multiprotocol Label Switching), IP 스위칭 등이 있다.

ATM은 라우터에서 패킷이 출력될 포트를 결정하기 위한 주소 검색을 필요로 하지 않는다. 즉, 각 패킷이 독립적으로 각 라우터에서 경로가 결정되어지는 것이 아니라, 호설정(Call signaling)단계에서 VPI/VCI 레이블을 이용하여 도착지까지 패킷이 지나갈 경로를 미리 설정하므로 패킷의 경로에 있는 라우터들에서의 주소 검색이 필요 없다. 한편 ATM은 셀의 크기를 53Bytes의 일정 크기로 규정하므로, 모든 패킷을 53Bytes로 자르는 조각화(Segmentation) 과정이 필요하다.

MPLS는 제 3계층 스위칭 기술의 일종으로써 제 2계층의 스위칭 교환 기술을 사용하여 고속으로 패킷을 전송하며, 제 3계층의 라우팅 기능을 사용하여 한 망의 확장성을 제공한다. 여기서는 패킷이 지나가야 하는 모든 라우터에서 IP 주소 검색을 해야 하는 기존의 방식과 달리 MPLS 네트워크에 진입하는 시점에 포워딩 검색을 단 한번만 수행함으로써, 검색 횟수를 최소화하고, 빠른 전송 속도를 갖는다.

IP 스위칭은 ATM과 IP 라우팅을 결합한 형태의 IP 스위치를 이용한다. IP 스위칭은 출발지와 도착지가 동일한 일련의 패킷들 중에서 첫 패킷에 대해서만 IP 라우팅을 적용하고, 나머지 모든 패킷들은 ATM 스위칭 방식으로 전송한다. 이러한 방식은 포워딩 과정의 병목인 주소 검색의 횟수를 감소시켜 빠른 전송 속도를 갖는다.

2.3 소프트웨어 기반 연구

라우터에서의 패킷 포워딩을 위한 주소 검색에 있어서의 소프트웨어적인 연구방향은 크게 두 가지로 나뉘어 진다. 그 첫째로, 주소 검색을 효율적으로 할 수 있는 알고리즘을 개발하는 것이고, 둘째는, 라우팅 정보를 압축하여 가능한 많은 정보를 접근속도가 빠른 캐시에 저장하여 데이터 접근에 의한 지연을 줄이는 방법을 개발하는 것이다.

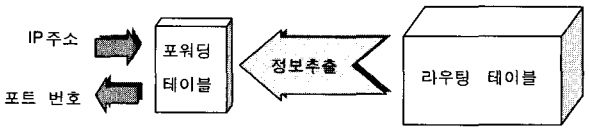
현재의 인터넷 환경에서 라우터가 라우팅 테이블을 보고 최종 목적지까지의 경로를 결정한다면, 매우 큰 라우팅 테이블을 직접 검색하여야 하므로 대단히 비효율적이다. 이에 현재는 라우팅 테이블의 정보 중 패킷이 전송되어져야 할 포트를 결정하는데 필요한 최소한의 정보인 프리픽스 값, 해당 프리픽스에 할당된 포트번호 등만을 포함한 포워딩 테이블을 만들어 이를 주소 검색에 사용하고 있다.

주소 검색 알고리즘에 대한 연구들은 다음과 같다. Patricia tries[9]는 이진 트라이의 각 노드 중에서 정보를 가지지 않고, 하나의 자식노드를 가지는 노드들을 제거함으로

써, 루트노드와 정보를 가지는 노드 간의 경로 상에서 의미 없는 노드들을 제거하는 경로 압축(Path compression)기법이다. 그러나 이 방법만으로는 LPM을 지원할 수 없으므로, 이후 여러 가지 방법들이 LPM을 지원하는 형태로 이를 발전시켜가고 있다. [16]은 라우팅 엔트리의 프리픽스 길이를 키로 하는 해시 테이블 기반의 이진 검색(Binary search)이다. 또한 검색이 이루어짐에 따라서 해시 테이블의 특정 엔트리에 보다 효과적인 검색을 할 수 있도록 특화되어지는 변화 이진 검색(mutating binary search)기법으로 보다 빠른 검색 방법을 제안하였다. [7]은 각 프리픽스의 정보를 IP 주소 전체 범위 내에서 그 시작점과 끝점을 가지는 하나의 범위정보로 변환하고, 이러한 범위정보들을 이진 검색 테이블(binary search table)에 정렬한다. 이것으로 검색대상이 속한 범위를 테이블에서 찾아내어, 대응하는 프리픽스 정보를 얻어내는 방식이다. 또한 6-way-search를 이용하여 시스템에 사용된 캐시 블럭의 크기를 고려한 검색 방식을 소개하였다. [15]에서는 동일한 정보를 가지는 프리픽스들을 묶어서, 보다 작은 길이의 프리픽스의 집합으로 변형시키고, 동적 프로그래밍(dynamic programing)에 기반한 최적화 기법을 사용하였다.

라우팅 테이블의 정보를 압축하는 방법에 관한 연구들은 다음과 같다. [11]은 이진 트라이에서 정보를 가지지 않고, 하나의 자식노드를 가지는 노드들을 제거하는 경로 압축기법과, 한 노드가 4개 이상의 자식 노드를 가질 수 있게 하여 단계를 줄이는 레벨 압축 기법을 함께 사용한 LC-trie를 제안하였다. [4]는 이진 트라이를 세단계로 구분하여 여러 개의 영역으로 나누고, 각 영역으로부터 구성된 비트 열을 16비트의 비트-마스크로 나눈다. 각 비트-마스크로부터 얻어지는 포워딩 정보의 포인터를 보다 빠르게 얻기 위하여, 추가적으로 code word array와 base index array를 사용하는 Lulea scheme를 제안하였다. Lulea scheme으로 구축된 포워딩 테이블의 크기는 매우 작지만, 영역의 개수와 포워딩 정보의 개수, 하위노드의 포인터 개수에 제한을 가진다. [5]에서는 [4]의 Lulea scheme에서 새로운 라우팅 엔트리가 발생하였을 때, leaf-pushing에 의한 포인터의 이전이 대량 발생할 수 있음을 지적하고, 확장 경로 비트맵(Extending Paths Bitmap)을 추가함으로써, 포워딩 테이블의 구축과 갱신을 쉽게 할 수 있게 만든 트리 비트맵(tree bitmap) 알고리즘을 제안하였고, [12]에서는 이진 트라이를 완전 이진 트라이로 변형한 뒤, 이를 여러 단계의 비트맵으로 구성하여, 비트맵의 1의 개수를 포워딩 정보의 인덱스로 사용하는 비트맵 트라이 검색 알고리즘을 제안하였다.

이와 같이 소프트웨어 기반의 연구는 입력된 패킷을 포워딩하기 위한 주소 검색 시간을 최소화 하기 위하여 라우팅 정보를 효율적으로 압축하여 저장하기 위한 데이터구조를 개발하거나 저장된 정보에 대하여 신속하게 검색을 수행 할 수 있는 알고리즘을 개발하는데 목적을 둔다. 그런데, 일반적으로 데이터 구조와 알고리즘을 개발하는 것은 서로 밀접하게 연관되어 있음에도 이들에 대한 연구가 독립적으로 이루어지는 사례는 거의 없다. 따라서 라우터의 검색 성능을 소프트웨어적인 방법으로 향상시키고자 하는 연구는 효율적인 알고리즘을 개발하는 종합적인 과정이라고 말할 수 있다.



(그림 1) 포워딩 테이블을 이용한 주소 검색

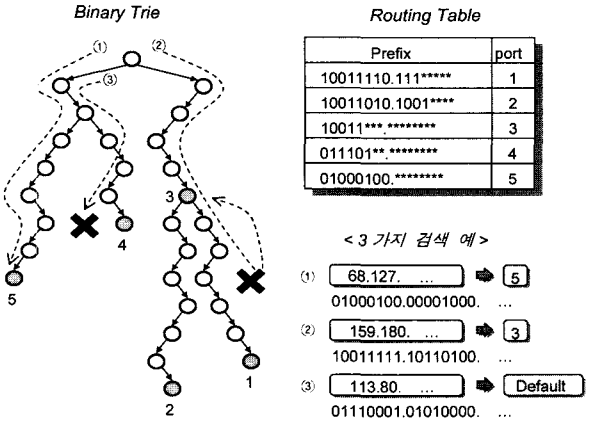
3. 비트맵 트라이

라우터에서 입력된 패킷의 출력포트를 결정하기 위한 주소 검색의 가장 고전적인 방법은 이진 트라이 구조의 포워딩 테이블에 대한 이진 검색이다.

이진 트라이 구조는 다음과 같은 방법으로 구축되어진다. 하나의 프리픽스가 가진 IP 주소는 최상위 비트부터 0인 경우 왼쪽, 1인 경우 오른쪽으로 자식노드를 추가해 나간다. 프리픽스 길이에 따른 노드의 추가가 종료되면 마지막에 생성된 터미널 노드에, 입력된 패킷이 출력될 포트번호 정보를 저장한다. 이때, 출력포트 정보를 가지는 노드를 원소 노드, 정보를 가지지 않는 노드를 비원소 노드라 부른다.

이진 트라이 포워딩 테이블의 검색에 사용되는 이진 검색에서는 LPM을 수행하여야 하므로, 더 이상 노드를 발견할 수 없을 때 까지 계속해서 트라이를 검색해 나가야 하며, 최종적으로 마지막에 검색된 원소 노드의 포트정보를 채택한다.

이진 트라이를 이용한 주소 검색에서는 다음 3가지 경우가 발생된다. 첫째, 원소 노드에서 검색이 종료되는 경우이다((그림 2)①). 이때 패킷은 터미널 원소 노드에 저장된 정보에 따라 포트를 할당받는다. 둘째, 검색 중 원소 노드를 거쳐 하위노드로 계속 검색이 진행되었으나 마지막에 비원소 노드에서 검색이 끝나는 경우이다((그림 2)②). 이 경우에는 이진 검색 경로의 마지막에 위치한 원소 노드에 저장된 정보가 사용된다. 이를 위하여 마지막으로 찾아낸 원소 노드에 대한 정보를 기억하여야 한다. 끝으로, 이진 검색 과정에서 원소 노드를 거치지 않는 경우이다((그림 2)③). 이때에는 디폴트 인터페이스 포트로 결정되어진다.



(그림 2) 이진 트라이 구조와 검색 예

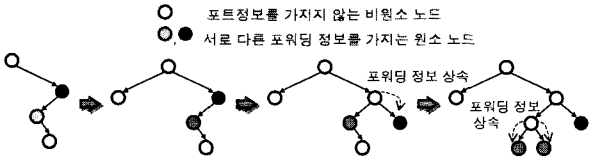
이와 같이 이진 트라이를 이용한 검색은 라우팅 테이블을 직접 검색하는 시간에 비하여 데이터 접근에 필요한 지연시간을 단축되어지며, 이진 트라이를 구축하는 작업이 쉽고 검색 알고리즘이 간단하다는 장점이 있다.

그러나 몇 만개의 프리픽스 정보를 가지는 이진 트라이 포워딩 테이블은 일반적인 컴퓨터의 캐시에 모두 저장될 정도로 작게 구성되어지지 않는다. 이것은 포워딩 테이블을 이진 트라이 구조로 구축할 때, 실제 정보를 가지는 원소 노드의 수에 비하여 라우팅 정보를 갖지 않는 비원소 노드의 수가 많아지기 때문이다. 예를 들어 본 연구에서 사용된 AADS[18] 테이블의 경우 원소 노드의 수는 33,858이고 비원소 노드의 수는 91,225이다. 이 경우, 이진 트라이에서의 원소 노드의 밀도는 27%에 불과하므로 포워딩 정보를 저장한다는 측면에서는 대단히 비효율적이다. 또한 보다 심각한 것은 수많은 비원소 노드를 접근 할 때 마다 데이터 접근에 따른 지연이 발생하여 포워딩 주소 검색에 오랜 시간이 소요된다. 결론적으로 말하자면, 이 방법은 공간적인 측면과 시간적인 측면 모두에서 문제를 안고 있다. 따라서 실제로 포워딩 정보를 저장하고 있지 않는 비원소 노드를 배제하고, 신속히 원소 노드를 찾아 포워딩 정보를 추출해 낼 수 있는 방법이 필요하다. 다음 절 에서는 본 논문에서 사용될 비트맵 트라이[12]의 기본적인 구조 및 그를 구성하는 방법과 그를 이용한 주소 검색 알고리즘을 간단히 소개하기로 한다.

3.1 비트맵 트라이의 구성

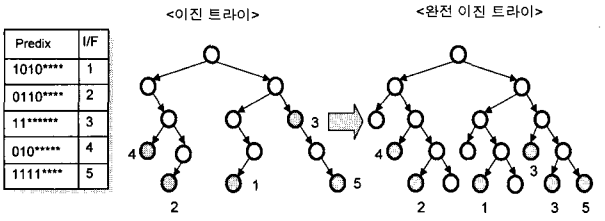
비트맵 트라이란 트라이 내에서의 원소 노드의 위치를 2개의 비트 열을 이용하여 나타내는 방식이다.

비트맵 트라이를 구축하기 위해서는 일차적으로, 포워딩 테이블로부터 구축된 이진 트라이를, 모든 노드가 자식노드를 0개 또는 2개를 갖도록 하는 완전 이진 트라이로 변환하여야 한다. 즉, (그림 3)과 같이, 포워딩 정보를 갖는 원소 노드의 자식이 0개 또는 2개라면 그대로 두고, 자식이 하나인 경우에는 부족한 자식을 추가하며, 자식노드(들)가 포워딩 정보를 갖지 않는 비원소 노드(들)라면 포워딩 정보를 자식(들)에게 상속해주고 자신은 비원소 노드로 남는다.



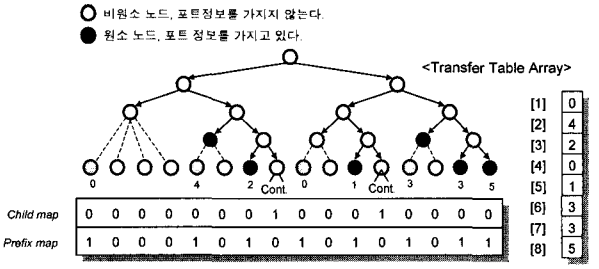
(그림 3) 완전 이진 트라이 확장 과정

이런 방식에 따라 구성된 완전 이진 트라이에서는 모든 원소 노드가 터미널 노드가 되는 특징을 갖는다(그림 4).



(그림 4) 완전 이진 트라이로의 변환

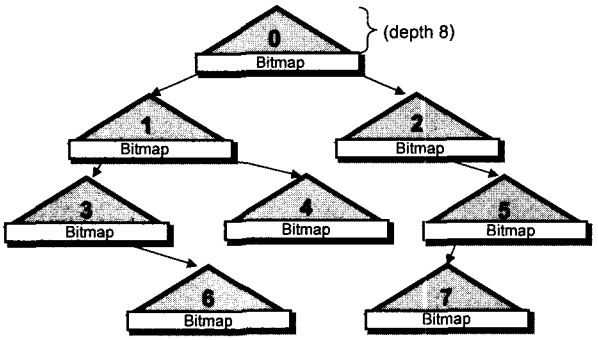
이제 완전 이진 트라이로부터 비트맵을 생성하는 과정은 다음과 같다. 완전 이진 트라이에 대하여 중위 순회(inorder traversal)[1]를 실행하는 도중 원소 노드를 만나면 비트 열에 '1'을 추가하고, 이 원소 노드에 저장된 포워딩 정보를 트랜스퍼테이블(transfer table)이라는 배열에 저장한다. 또한 이진 트라이의 최하단 노드의 높이(H)를 0이라 할 때, 원소 노드의 높이가 1 이상인 경우 '1' 뒤에 2H-1개의 '0'을 추가한다. 이와 같은 방식으로 생성된 비트 열을 Prefix-map (또는 P-map)이라 한다(그림 5).



(그림 5) Prefix-Map과 Child-Map, Transfer Table 구성의 예

IPv4의 주소체계는 32비트로 구성되어있기 때문에, (완전)이진 트라이는 최대 높이가 32가 되고 최하위에 위치할 수 있는 노드의 수는 최대 232개이다. 따라서 주어진 라우팅 테이블로부터 P-map을 생성하면 512Mbytes 크기의 비트 열이 구성된다. 그러나 실제로 사용되는 라우터의 라우

팅 테이블의 엔트리 수를 최대 10만개까지 고려하더라도 생성되는 비트 열에서 1의 수를 가지는 부분은 극히 일부에 지나지 않는다. 이진 트라이에서 노드가 존재하는 곳만을 나누어 비트 열을 생성시키게 되면, 정보를 가지지 않는 대량의 비트 열은 생성되지 않는다. 노드들을 깊이 8로 나누어 각 이진 트라이들에 대해 위와 같은 방법으로 P-map를 생성하면 각 비트 열은 256비트이고, 모두 네 단계의 서브 비트맵 트라이로 비트 열을 구성한다. (그림 6)은 이러한 서브이진 트라이로 구성된 비트열의 개념도 이다.

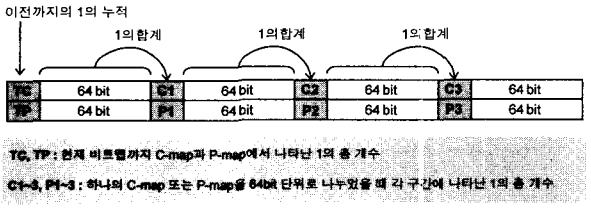


(그림 6) 높이 8의 서브이진 트라이들로 구성된 네단계의 비트맵 트라이 구조

각 서브이진 트라이들로부터 생성된 256비트 크기의 P-map들은 최상위 P-map으로부터 전위순회 순서로 순차적으로 연결하여 최종적인 P-map의 배열을 구성한다.

한편, 각 서브이진 트라이의 최 하단에 있는 비원소 노드 중에서 자식을 가진 노드에 대해서는 '1'을 배정하고, 그 밖의 모든 노드에는 '0'을 배정한 또 다른 비트맵을 만들어 Child-map(또는 C-map)이라 부른다(그림 5). 비트맵이란 이상에서 생성된 P-map과 C-map을 아우르는 명칭이다.

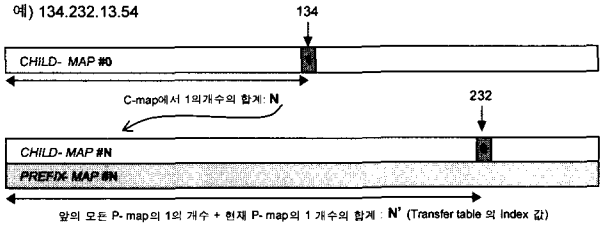
끝으로, 보다 빠른 검색을 위하여, 각 비트맵에는 (그림 7)과 같이 그 비트맵에 도달하기까지의 1의 개수의 누적과, 64비트씩 4구간으로 나누어 각 구간의 1의 값을 합산한 구역합계를 함께 저장한다.



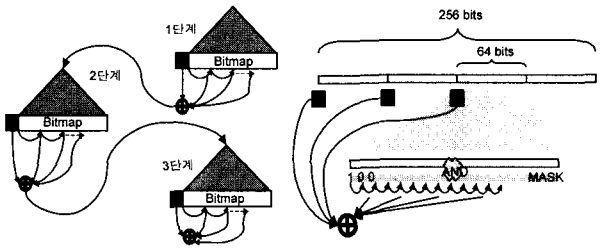
(그림 7) 하나의 서브이진 트라이를 위한 비트맵의 테이터 구조

3.2 비트맵 트라이 검색 알고리즘

패킷의 목적지 주소의 첫 8비트의 값이 η일 때, 최상위 C-map의 X번째 비트의 값이 1이면, 이 C-map의 처음부터 η번째 비트까지의 1의 개수가 다음에 검사해야 할 하위 C-map의 인덱스가 된다. 선택된 하위 C-map에서 패킷의 목적지 주소의 다음 8비트 값에 대하여 같은 방식으로 검사가 이루어진다. C-map의 해당 비트가 0인 경우, 즉 터미널 노드이면, 최상위 P-map으로부터 현재 P-map의 해당 비트까지의 1의 개수를 트랜스퍼 테이블의 인덱스 값으로 사용한다. 그러면 트랜스퍼 테이블로부터 해당 프리픽스의 포트 정보를 얻을 수 있다. (그림 8), (그림 9) 자세한 알고리즘은 [12]에서 찾을 수 있다.



(그림 8) 비트맵 검색의 예



(그림 9) 비트맵 트라이의 검색 과정

4. 향상된 비트맵 트라이

본 논문의 제3장에서 논한 비트맵 구조의 포워딩 테이블을 이용한 주소 검색 방법은 포워딩 테이블의 크기를 76% 이상 감축하였다. 이 정도의 크기는 비트맵의 대부분을 캐시에 저장할 수 있어 이진 트라이에서의 메모리 접근을 95%이상 감소시키는 효과를 얻을 수 있었다. 〈표 1〉 그러나, 이 알고리즘은 3가지 측면에서 개선되어야 할 여지를 안고 있다. 다음의 절 들에서는 이 문제점을 간단히 설명하고, 이들을 해결하기 위하여 본 연구에서 제안하는 방안들에 대하여 자세히 논하기로 한다.

4.1 카운트 테이블을 이용한 시프트 연산의 감축

기존의 비트맵 트라이 알고리즘에서는 두개의 비트맵으로부터 포워딩 정보가 저장되어있는 트랜스퍼 테이블의 인덱스를 얻기 위해서, P-map과 C-map에서의 1의 개수를 세어야 하는데, 이를 위한 많은 시프트 연산이 필요하다. 이와 같은 시프트 연산의 수는 <표 4>에서 볼 수 있다. <표 1>과 <표 3>에서 보듯이 통상적인 이진 트라이에 비해 메모리 요구량은 적지만 시프트 연산으로 인한 CPU 내에서

의 명령어 처리지연이 급증함으로써 검색 시간은 오히려 증가하는 결과를 초래한다.

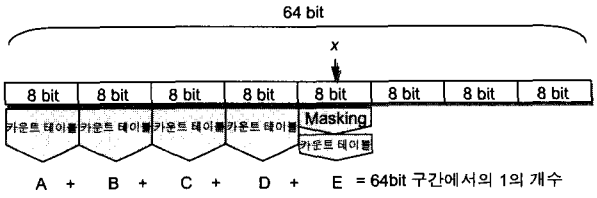
〈표 1〉 이진 트라이와 비트맵 트라이 알고리즘의 지연 요소 비교

	메모리요구량	명령어 수	캐시접근 수	메모리접근 수
이진 트라이 알고리즘	2,001KB	146,637,431	25,187,076	163,002
비트맵 트라이 알고리즘	469KB	472,048,634	11,812,000	7,214

본 연구에서 제안하는 향상된 알고리즘에서는, 이러한 시프트연산을 제거하는 방법으로 8비트의 이진 숫자로 표현할 수 있는 모든 256경우마다의 1의 개수를 미리 세어 「카운트 테이블」이라 불리는 새로운 공간에 저장하고, 이를 사용하여 8비트 숫자에서의 1의 개수를 구하는 방법을 도입하였다.

카운트 테이블을 이용하여 1의 개수를 얻는 방법은 다음과 같다. (그림 10)(a)와 같이 검색할 비트열의 이진수 값이 00010110, 즉 십진수로 22일 경우에 비트열의 1의 개수를 구하기 위해 시프트 연산을 사용하는 대신 비트열의 값을 그대로 카운트 테이블의 인덱스로 사용하여 카운트 테이블의 22번째 값인 3을 가져옴으로써 1의 개수를 얻는다. 만일 (그림 10)(b)처럼, x위치까지의 1의 개수를 구할 경우에는, x위치 이후의 비트들은 마스크를 통하여 0이 되어지고, 이를 통해 나온 비트열의 값인 20을 카운트 테이블에 인덱스로 사용한다.

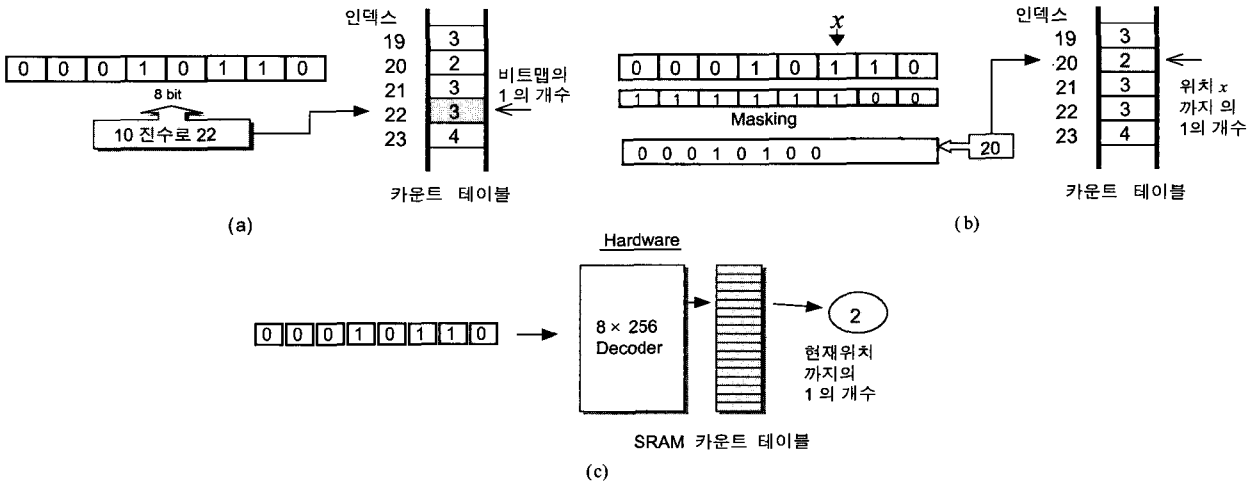
(그림 7)에서 본 바와 같이, 각 비트맵은 64비트의 구역으로 나뉘어 중간합계를 가지고 있으므로, 64bit의 구역을 8비트로 나누어 각각 1의 개수를 구하되, (그림 11)에서 보듯이 앞쪽의 8비트 단위의 비트 열들은 카운트 테이블로부터 바로 1의 개수를 얻어내고, x위치의 비트 열은 마스킹한 이후에 카운트 테이블로부터 1의 개수를 얻어낸다. 이렇게 얻어진 각 8비트 단위의 1의 개수를 합산하여, 해당 64비트 구역에서의 1의 개수를 구한다. 카운트 테이블의 크기는 256Byte에 불과하므로 (그림 10)(c)와 같이 하드웨어로 구현하면 보다 빠른 검색을 달성할 수 있다.



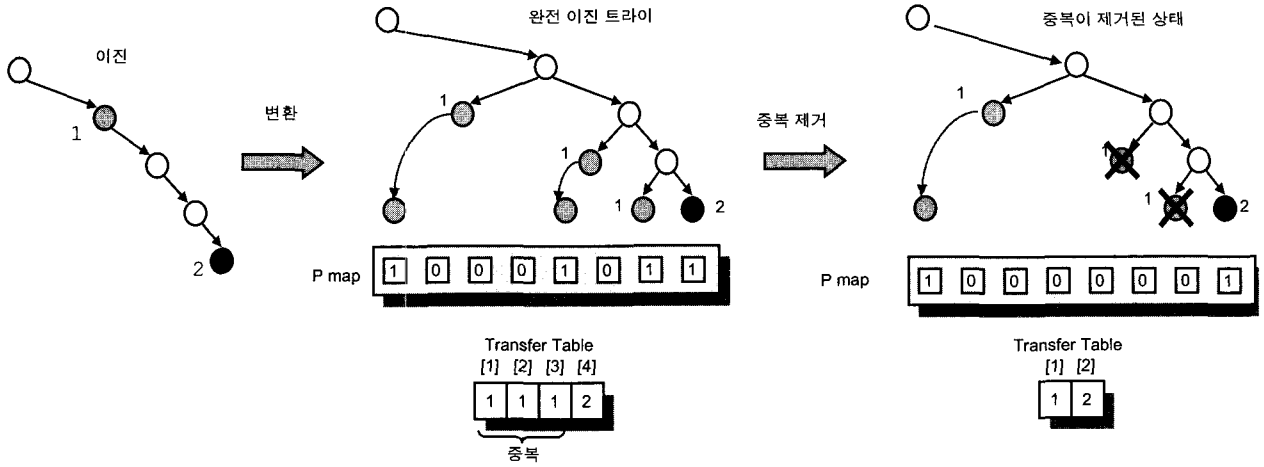
(그림 11) 64bit 구역에서 카운트 테이블을 이용한 1의 개수 구하기

4.2 트랜스퍼 테이블의 중복된 정보 제거

라우팅 테이블로부터 두개의 비트맵을 구축하기 위해서는 제 3.1절에서 설명된 바와 같이 이진 트라이를 완전 이진 트라이로 변환하는 과정이 선행되어야 한다. 이 과정에서 상속에 의하여 동일한 포워딩 정보를 갖는 원소 노드의 수가 증가하게 되며 이는 트랜스퍼 테이블에 동일한 정보를 중복하여 저장하는 결과를 낳는다. 이는 곧, 트랜스퍼 테이블의 메모리 요구량을 증가시킨다. 한편, 비트맵 구조의 포워딩 테이블을 구축하는 가장 큰 목적은 테이블이 캐시에 모두 저장될 정도로 압축시키는 것이다. 그런데 트랜스퍼 테이블의 크기가 커짐에 따라 본래의 목적을 달성하는데 장애요소가 될 우려가 있다. 실제로 본 연구에서 사용한 라우팅 테이블로부터 생성된 트랜스퍼 테이블은 그 크기가 94Kbyte에 이른다. 이는, <표 1>에서 보여주는 비트맵의 메모리 요구량에 비교하였을 때 20%를 상회하는 크



(그림 10) 카운트 테이블의 사용



(그림 12) 트랜스퍼 테이블의 중복된 정보의 제거 예

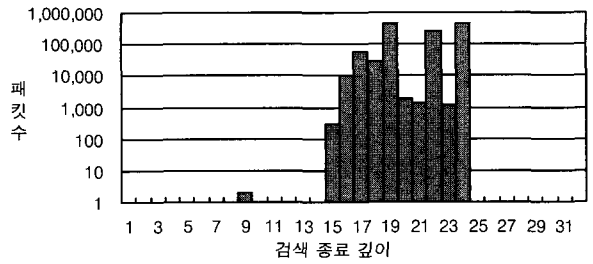
기이므로 포워딩 주소 검색을 위한 모든 데이터를 캐시에 저장하는데 커다란 영향을 준다.

본 연구에서는 이러한 중복된 정보를 제거하기 위하여, 이진 트라이에 대하여 중위 순회를 실행하며 트랜스퍼 테이블을 구축할 때, 최근의 라우팅 정보와 동일한 라우팅 정보를 갖는 원소 노드를 발견하면 이의 정보는 테이블에 저장하지 않으며, P-map에도 1을 발생시키지 않는 방안을 채택하였다(그림 12). 이 방식에 따라 생성된 트랜스퍼 테이블은 52Kbytes에 불과하여, 원래의 트랜스퍼 테이블의 크기를 45% 정도 감소하는 효과를 얻었다.

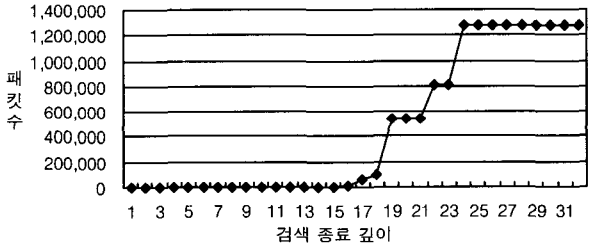
4.3 IP 주소 검색범위의 다원화

비트맵 트라이의 포워딩 정보는 그 위치에 따라 접근빈도에 상당한 편차가 있다. 즉, 비트맵들의 상위레벨에 해당하는 위치의 비트정보는 거의 매 패킷마다 사용되므로 빈번하게 접근되지만, 하위레벨에 해상하는 위치의 비트정보는 사용 빈도가 상대적으로 낮다. 기존의 비트맵 트라이 알고리즘에서는 사용 빈도가 낮은 비트맵의 데이터 구조를 중위의 데이터 구조와 동일하게하여, 향상된 성능을 얻을 수 있는 요소를 간과하였다.

(그림 13)은 동국대학교 라우팅 테이블로 구축한 이진 트라이에 동국대학교 패킷 트레이스로 포워딩 실험 하였을 때 나타난 결과이다. 동국대학교 라우팅 테이블에서 가장 짧은 길이의 프리픽스는 12이며, 약 50% 프리픽스가 24의 프리픽스 길이를 가지는 전형적인 Class C의 형태를 가진다. 위에서 나타나는 것과 같이, 대부분 깊이 16에서 24사이 노드에서 검색이 종료되고, 매우 적은수의 패킷만이 24 노드 이후에 검색이 종료되는 분포를 보인다. 트라이의 16 이상 상위 레벨의 노드는 거의 모든 패킷에 대한 검색이 이루어질 때마다 접근이 이루어진다.



(그림 13) 노드 깊이에 따른 검색종료 위치 분포도



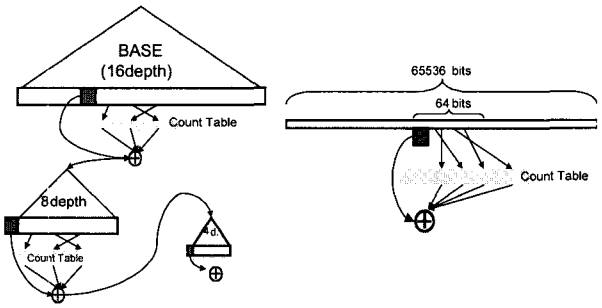
(그림 14) 노드 깊이에 따른 검색종료 위치 누적 분포도

이러한 특성을 비추어 보아, 깊이 24까지를 상위 비트맵으로 구성하면 서브이진 트라이의 깊이를 8로 했을때 보다 비트맵들에 대한 데이터 접근의 수를 감소시켜, 검색속도를 개선할 수 있다. 그러나 이 경우 상위 비트맵의 크기는 약 9Mbyte에 달하게 되고, 이 경우 캐시 접근실패에 따르는 메모리 접근 지연이 급증하여 심각한 성능저하가 야기된다. 그러므로 상위 비트맵을 구성하는 깊이를 16으로 하면 필요한 메모리 요구량은 20kbyte이며, 깊이 8의 중위 비트맵을 사용하도록 하였다. 이렇게 하면 비트맵을 한번 더 거쳐야 하기 때문에 데이터 접근수가 증가하지만, 포워딩 테이블이 작게 구성되어지고 캐시 접근 성공률이 증가하여 메모리 접근 지연은 감소한다. 캐시 접근 실패에 따른 메모리 접근지연은 캐시 접근 지연에 비하여 수십 배의 시간을 소

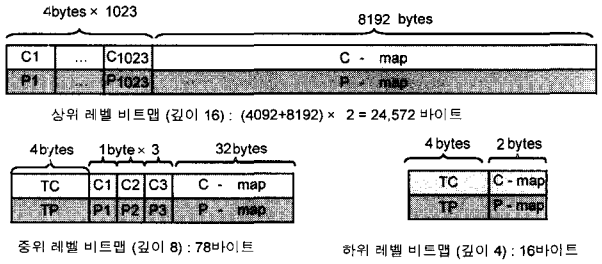
모하므로 전체적인 성능의 향상이 이루어진다.

또한 깊이 24이후의 검색은 거의 이루어지지 않는다. 이러한 하위 비트맵들은 그 수에 비해 접근 빈도가 극히 낮으므로 비트맵을 구성하는 원리에 있어서 검색 속도의 개선보다는 메모리 요구량에 초점을 맞추는 것이 더 효과적이다. 때문에, 기존 비트맵이 사용한 깊이 8의 비트맵을 두 단계의 하위 비트맵으로 나누어 깊이 4 비트맵을 구성함으로써 전체 포워딩 테이블의 크기를 줄이도록 하였다.

이에 본 연구에서는 비트맵을 16-8-4-4의 깊이로 구분하여 구성하였다(그림 15). 상위 16레벨로 구성된 비트맵의 데이터 구조를 기존의 비트맵처럼 64비트 영역으로 나누어 1의 개수에 대한 중간합계를 저장하면, 검색과정에서 최대 1024번의 ADD연산이 발생한다. 때문에 각 중간합계를 누적합계로 대체하여 한번의 데이터 접근으로 1의 개수를 얻도록 하였다. 또한 하위의 4레벨 비트맵을 구성하는 비트열의 크기가 16비트 이므로 중간 합계를 가지지 않으며, 빠른 속도로 1의 개수를 얻을 수 있다(그림 16).



(그림 15) 검색 범위의 다원화



(그림 16) 향상된 비트맵 트라이의 3가지 비트맵 구조

4.4 향상된 비트맵 트라이 주소 검색 알고리즘

본 연구에서 제안하는 알고리즘은 (그림 16)과 같으며 이을 간단히 설명하면 다음과 같다. 패킷이 들어오면 목적지 주소의 상위 16비트를 얻는다(줄 1). 그리고 최 상위 비트맵에서 이 16비트 값의 십진수 값 위치에 있는 C-map의 비트를 검사한다(줄 3). 이때 그 비트가 1이면 다른 비트맵에 정보가 있으므로 C-map에서 위의 16비트 값 위치까지의 1의 개수를 구하여 다음 비트맵의 인덱스를 구하고(줄 4~6) 0이면 현재 비트맵의 P-map에서 1의 개수를 구하여 (줄 7~9) 트랜스퍼 테이블의 인덱스를 얻는다(줄 10). 최 상위 비트맵에서 검색이 종료되지 않게 되면 중위 비트맵으로 검색이 옮겨오게 된다. 중위 비트맵에는 목적지 주소의 다음 8비트의 값을 가져와(줄 11) 얻어진 8비트 값의 십진수 값 위치에 있는 C-map의 비트를 검사한다(줄 13). 최 상위 비트맵과 마찬가지로, 검사된 비트가 1인 경우 하위 비트맵의 인덱스를 구하고(줄 14~16) 그렇지 않으면 트랜스퍼 테이블의 인덱스를 구하는 것으로 검색이 종료된다(줄 17~20). 여기까지가 상위 깊이 24의 검색이며, (그림 13, 14)에서 본 바와 같이 대부분의 검색은 중위 비트맵에서 종료되어진다. 그러나 만일 중위 비트맵에서 종료가 끝나지 않아 하위 비트맵 인덱스를 구하게 된 경우, 패킷 목적지 주소의 다음 4비트의 값으로(줄 22) 하위 비트맵을 검사하게 된다. 먼저 위의 4비트 값의 십진수 값 위치에 있는 C-map의 비트를 검사하고(줄 23), 그 비트가 0이면 트랜스퍼 테이블 인덱스를 구하는 것으로 검사가 종료되어진다(줄 27~30). 하위 비트맵은 두 단계로 구분되어 있으므로 첫 하위 비트맵에서 검색이 종료되지 않으면 최종적으로 하위 비트맵에서 검색이 종료되어 지는데(줄 32~36), 두 번째 하위 비트맵까지 검색이 진행되는 경우, 마지막 목적지 주소 4비트 값으로 C-map을 검사할 필요는 없다. 하위 비트맵들은 상, 중위 비트맵에 비하여 데이터 구조가 간결하기 때문에 보다 빠르게 검색이 이루어질 수 있다. 이렇게 최대 네 단계를 거쳐 얻어진 인덱스를 사용하여 트랜스퍼 테이블로부터 출력 포트 번호를 얻어내는 것으로 패킷의 포워딩 결정이 이루어진다.

```
1 :  get addr = IP_Address[0~15] ; BM_node = Base_Bitmap ;
2 :  get byte_idx = addr >> 4 ; region_idx = byte_idx >> 4 ;
     bit_idx = addr & 0xF ;
3 :    if (BM_node's c-bitmap's byte(byte_idx) & (0x80 >>
       bit_idx))
4 :      bm_idx = c[region_idx] ;
5 :      for (i = (region_idx<<4) to byte_idx-1) bm_idx += CT
       [c-map(i)] ;
6 :      bm_idx += CT[ c-map(byte_idx) & MT[bit_idx] ] ;
7 :    else transfer_idx = p[region_idx] ;
8 :      for (i = (region_idx<<4) to byte_idx-1) transfer_idx +=
       CT[p-map(i)] ;
9 :      transfer_idx += CT[p-map(byte_idx) & MT[bit_idx] ] ;
10 :     return transfer_table[transfer_idx] ;
11 : get addr = IP_Address[16~23] ; BM_node = BM(bm_idx) ;
12 : get byte_idx = addr >> 4 ; region_idx = byte_idx >> 4 ;
     bit_idx = addr & 0xF ;
13 :   if (BM_node's c-bitmap's byte(byte_idx) & (0x80 >>
       bit_idx))
14 :     bm_idx = tc + c[region_idx] ;
```

```
15 :        for (i = (region_idx<<4) to byte_idx-1) bm_idx +=
            CT[c-map(i)] ;
16 :        bm_idx += CT[ c-map(byte_idx) & MT[bit_idx] ] ;
17 :    else transfer_idx = tp + p[region_idx] ;
18 :        for (i = (region_idx<<4) to byte_idx-1) transfer_idx +=
            CT[p-map(i)] ;
19 :        transfer_idx += CT[ p-map(byte_idx) & MT[bit_idx] ] ;
20 :        return transfer_table[transfer_idx] ;
21 :  get addr = IP_Address[24~27] ; BM_node = BM(bm_idx) ;
22 :  get bit_idx = addr & 0xF ;
23 :    if (BM_node's c-bitmap & (0x8000 >> bit_idx))
24 :        bm_idx = tc ;
25 :        if (addr < 8) bm_idx += CT[c-map(0) & MT[bit_idx]] ;
26 :        else bm_idx += CT[c-map(0)] + CT[c-map(1) & MT
            [bit_idx]] ;
27 :    else transfer_idx = tp ;
28 :        if (addr < 8) transfer_idx += CT[p-map(0) &
            MT[bit_idx]] ;
29 :        else transfer_idx += CT[p-map(0) + CT[p-map(1) &
            MT[bit_idx]] ;
30 :        return transfer_table[transfer_idx] ;
31 :  get addr = IP_Address[28~31] ; BM_node = BM(bm_idx) ;
32 :  get bit_idx = addr & 0xF ;
33 :    transfer_idx = tp ;
34 :    if (addr < 8) transfer_idx += CT[p-map(0) & MT[bit_idx]] ;
35 :    else transfer_idx += CT[p-map(0) + CT[p-map(1) & MT
        [bit_idx]] ;
36 :    return transfer_table[transfer_idx] ;
```

BM : 비트맵 CT : 카운트 테이블 MT : 마스크

(그림 17) 향상된 비트맵 트라이 검색 알고리즘

5. 시뮬레이션 환경

5.1 실행-구동 시뮬레이션의 필요성

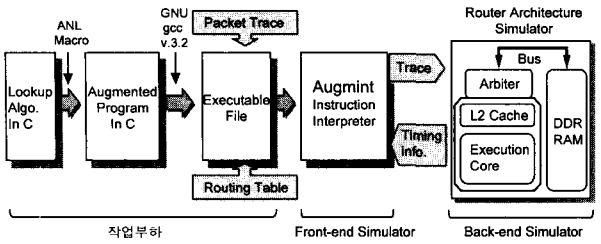
알고리즘의 성능을 표현하는 대표적인 방법은 그 알고리즘의 복잡도를 O(big-oh) 기호를 사용하는 것이다. 이는 주어진 알고리즘을 실행하는데 소요되는 명령어의 수를 나타낸다[1]. 하지만, 이러한 해석은 알고리즘이 실행되고 있는 시스템에 무관하게 표현한 것으로써 주어진 알고리즘의 정확한 실행시간을 표현하기에는 부정확하다. 왜냐하면 프로그램이 실행될 때의 명령어중 상당한 부분이 데이터를 접근하는 명령어이기 때문이다. 데이터 접근 명령어는 대부분이 CPU내에서 처리되고 명령어에 비하여 보다 많은 명령어 사이클을 필요로 한다. 더욱이 메모리에 저장되어 있는 데이터를 접근하기 위해서는 시스템 버스를 통하여 통상 32~64바이트에 이르는 한 캐시 블록을 전송해야 하므로 소요되는 시간이 더욱 길어진다. 우수한 알고리즘을 개발하기 위해서는 명령어의 수, 캐시에 저장되어있는 데이터에 접근하는 수, 메모리에 저장되어있는 데이터에 접근하는 수를 명확히 구별하여 측정하고, 각 요소의 정확한 평가와 분석을 통한 개선을 바탕으로 종합적으로 향상시킬 수 있는 도구가 필수적이다.

이에, 본 논문에서는 라우터의 성능에 직접적으로 영향을 미치는 지연요소를 명령어 처리지연, 캐시 접근지연, 메모리 접근지연으로 구분하고 이들 각 지연 요소를 최소화하기 위한 방안을 모색한다. 명령어 처리지연이란 프로세서가 주소 검색을 위하여 일련의 명령어를 순차적으로 또는 반복적으로 수행할 때, 메모리나 캐시로부터 데이터를 가져오는 시간을 제외한 순수한 처리 지연시간이다. 캐시 접근지연이란 명령어들이 프로세서 내부의 레지스터가 아닌 프로세서에 가장 인접한 저장장치인 캐시에 저장된 데이터를 접근하는데 필요한 지연이다. 메모리 접근 지연이란, 요구하는 데이터가 캐시에 없을 때 캐시 접근실패가 발생하여 주 메모리로부터 데이터를 가져오는데 발생하는 지연시간을 말한다.

5.2 실행-구동 시뮬레이션 환경

본 연구에서는 실행-구동 시뮬레이션 기법으로 실험을 진행하였다. 이를 위해서는 명령어 해석기(instruction interpreter) 또는 명령어 집합 시뮬레이터(instruction set simulator)가 필수적이다. 이는 특정한 마이크로프로세서를 그대로 소프트웨어로 구현하고 가상 메모리 모듈 역시 소프트웨어로 구현한 도구이다. 이 도구는 작업부하(workload)인 컴파일 된 실행 가능한 파일을 명령어 줄 단위로 읽어들여 마치 일반 컴퓨터에서 실행하는 것과 동일한 방법으로 실행하는 기능을 갖는다. 일반적으로, 설계된 알고리즘에 따라 프로그램으로 구현된 작업부하는 명령어 해석기가 적절하게 해석할 수 있도록 매크로(macro)라 불리는 특수 명령어가 추가(augment)된다. 라우터 시스템의 실험을 위한 작업부하에는 라우팅 테이블과 라우터에 입력되는 패킷 트레이스가 포함된다.

본 연구에서 사용된 명령어 해석기는 Unicersity of Illinois에서 개발된 Pentium 프로세서를 모델링한 개방 소프트웨어인 AugMint[14]를 사용하였다. 이는 Intel사의 Pentium-4 프로세서와 Linux Red-Hat 2.4.18 버전의 플랫폼에서 실행되었다. 작업부하인 프로그램은 C언어로 구현되었으며 ANL 매크로를 추가하여 GNU gcc v.32에 -O1옵션을 사용하여 컴파일함으로써 그 실행 파일을 AugMint가 해석할 수 있도록 하였다. 비트맵을 구성하는데 사용된 라우팅 테이블은 33,858개의 엔트리를 가지는 AADS 테이블[18]을 사용하였다. 패킷 트레이스(packet trace)는 동국대학교에서 수집된 것으로써 약 120만개의 패킷을 포함하며 본 연구에서는 각 패킷의 목적지 IP 주소를 이용하여 실험을 수행하였다.



(그림 18) 시뮬레이션 환경

명령어 해석기는 매 명령어를 실행할 때마다 CPU안에서 발생하는 사건들을 출력하는데 이를 실행 트레이스라 부른다. 트레이스가 포함하는 정보 중에서 일반적으로 가장 많이 쓰이는 것은 프로세서 번호(PID), 명령어, 그리고 명령어가 데이터 접근 명령어인 경우에는 데이터의 가상 주소 등이 있다(단, 프로세서 번호는 다중 프로세서를 시뮬레이션 할 때 유용하게 사용되는데 본 연구에서는 단일 프로세서 모델을 다루므로 사용되지 않는다.). 따라서 실행-구동 시뮬레이션 기법의 실험을 수행하면 연구의 대상이 되는 알고리즘을 구현한 프로그램의 모든 실행 정보를 정확하게 파악 할 수 있다.

특히 트레이스에 포함된 데이터 접근정보는 캐시나 메모리에서 발생하는 사건을 시뮬레이션하기 위하여 라우터 아키텍처 시뮬레이터에 전달된다. 이는 특정한 캐시와 주 메모리 모델을 구현한 소프트웨어이다. 트레이스 정보에 포함된 데이터 접근 유형(읽기 또는 쓰기)과 데이터의 가상 주소를 이용하여 캐시 내에서의 상태 정보를 변화시키며 그 상태와 데이터 접근 유형에 따라 데이터를 캐시와 메모리 사이를 PCI버스를 경과하여 이동시킨다. 따라서 이로부터 캐시 접근 성공(cache Hit) 또는 실패(cache miss)의 수, 캐시로부터 메모리로의 데이터 이동(write-Back 또는 write-through) 횟수 및 메모리로부터 캐시로의 데이터 이동(block fetch)의 횟수 등을 알 수 있다.

여기서 무엇보다 중요한 것은 라우터 아키텍처 시뮬레이터는 트레이스에 포함된 데이터 접근 정보에 따른 사건이 처리되는데 필요한 지연시간을 계산한다. 이 시간 정보는 명령어 해석기에 제공되며, 명령어 해석기는 실행 가능 파일의 다음 명령어 줄을 읽어 들이기 전에 이 지연 시간만큼 실행을 지연시킨다. 결론적으로, 실행-구동 시뮬레이션을 수행하면, CPU 내부에서 실행되는 명령어의 수와 이로 인한 지연시간, 캐시접근 성공 횟수와 그로 인한 지연시간과 캐시접근 실패 횟수와 그로 인한 지연시간 및 최종적으로 전체적인 실행시간을 구할 수 있다.

본 연구에서는 2.53㎓의 Pentium-4 프로세서와 이에 부수적으로 사용되는 캐시모델을 사용하였다. 즉, 캐시의 크기는 512KBytes로 하고 64Byte 블록을 8-way set associative 방식으로 맵핑하였다. 그리고 교체(Replacement)정책은 LRU(Least Recently Used)을, 되쓰기 정책은 Write-Back 방식을 가정하였다. 또한, DDR 2100 SDRAM을 기준으로 하여 266㎒ 메모리와 64비트 데이터 라인을 갖는 133㎒ PCI버스를 가정하였다.

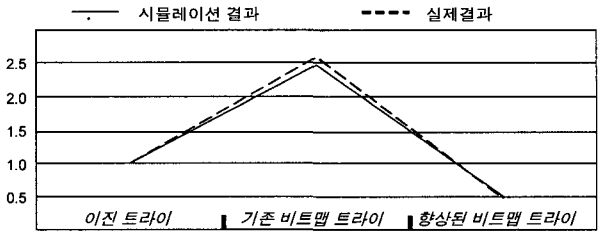
6. 시뮬레이션 결과

시뮬레이션의 가장 큰 문제점 중의 하나가 시뮬레이션을 통한 측정결과가 실제 시스템에서의 실측결과와 상이할 수 있다는 사실이다. 이는 시뮬레이터나 작업부하를 모델링할 때 시스템 등의 변수 값이 실제의 값과 차이가 있기 때문이다. 따라서 시뮬레이션을 통한 결과의 신뢰도, 나아가 연구의 질을 높이기 위해서는 정확한 모델링과 더불어 실행-구동 시뮬레이션과 같은 높은 신뢰도의 실험 기법을 사용하는 것이 무엇보다 중요하다.

본 연구에서는, 최우선적으로, 시뮬레이션 결과의 신뢰성을 확보하고자 하였다. 즉, 이진 트라이, 기존 비트맵 트라이 및 향상된 비트맵 트라이 등 세 가지 알고리즘의 실행시간을 시뮬레이션 결과와 실제의 시스템으로부터 실측한 결과를 비교하여 보았다. <표 2>와 (그림 19)는 이들 세 가지 알고리즘들의 실행시간을 이진 트라이를 이용한 검색 알고리즘을 기준으로 하여 정규화한 값이다. <표 2>에서 볼 수 있듯이 본 연구에서 채택한 시뮬레이션 기법과 시스템 및 작업부하는 실측한 결과에 매우 근접한 결과를 제공하며, 따라서 본 논문에서 제시하는 연구 결과에 높은 신뢰성을 부여할 수 있음을 알 수 있다.

〈표 2〉 실시간 측정치와 시뮬레이션 측정치의 비교

	이진 트라이	기존 비트맵 트라이	향상된 비트맵 트라이
시뮬레이션 (명령어 사이클 수)	181,574,061	461,088,286	83,673,152
정규화 된 값	1	2.539	0.461
실측 시간(㎲)	173,828	426,484	83,984
정규화 된 값	1	2.453	0.483



(그림 19) 실제결과와 시뮬레이션 결과의 정규화수치 비교

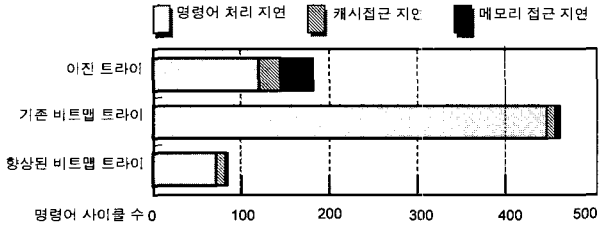
각 알고리즘의 실행시간을 명령어 처리지연, 캐시 접근

성공에 따른 캐시 접근지연, 캐시 접근 실패에 따른 메모리 접근지연의 세 가지 요소로 구분하여 비교하면 <표 3>과 (그림 20)과 같다.

〈표 4〉 각 알고리즘의 검색시간 비교

(단위 : 명령어 사이클 수)

	이진 트라이	기존 비트맵 트라이	향상된 비트맵 트라이
명령어 처리 지연	119,228,457	447,631,494	70,998,239
캐시 접근 지연	25,187,076	11,812,000	11,139,333
메모리 접근 지연	37,158,528	1,644,792	1,535,580
합 계	181,574,061	461,088,286	83,673,152



(그림 20) 각 알고리즘의 명령어 사이클 수 비교

(그림 20)에서 보면 기존의 비트맵 트라이 알고리즘의 명령어 처리지연은 이진 트라이 알고리즘의 그것보다 3배가량 많다. 이는 <표 4>에서 보는 바와 같이 너무 많은 수의 시프트 연산으로 인한 것임을 이미 밝혔었다. 즉, 기존의 비트맵 트라이 알고리즘에서는 라우팅 정보를 압축하여 이진 트라이에 대비하여 각각 47%와 4%에 불과한 캐시 접근지연과 메모리 접근지연을 달성했음에도 불구하고 시프트 연산으로 인하여 오히려 전체적인 실행시간이 2.5배 이상 증가하는 결과가 초래되었다.

한편, 본 연구에서 제시한 향상된 비트맵 트라이 알고리즘은 이진 트라이 알고리즘과 기존의 비트맵 알고리즘에 비교하여 각각 59%, 18%에 불과한 명령어 처리지연을 나타낸다. 이는 본 연구에서 사용한 카운트 테이블을 이용할 경우 시프트 연산을 제거한 결과이다. 또한, 비트맵 트라이의 검색범위를 16-8-4-4로 다원화하여 대부분의 주소 검색이 최상위 레벨의 단일 서브비트맵 트라이 안에서 이루어지도록 하는 반면 하위 레벨의 서브비트맵 트라이의 크기를 줄여 필요이상의 하위 레벨의 노드에 대한 검색을 근본적으로 배제하여 얻을 수 있는 효과이다. 한편 캐시 접근지연과 메모리 접근지연은 이진 트라이 알고리즘에 비교해서는 탁월한 성능을 보여준다. 또한 기존의 비트맵 트라이 알고리즘보다는 약간의 성능 개선효과를 보여주는데 이는 트랜스퍼 테이블의 크기를 줄임으로써 보다 많은 유용한 데이터를 캐시에 저장함으로써 얻어진 결과이다.

〈표 5〉 검색 알고리즘의 처리 지연 요소 비교(명령어 수)

구 분	지 연 요 소	지연	평균 단계	합계
이진 트라이 검색 알고리즘	원소 노드인가 확인	1	× 24	120
	Bit추출	1		
	Bit비교 (1 or 0)	1		
	포인터 얻어오기	1		
	노드 불러오기	1		
기존 비트맵 트라이 검색 알고리즘	비트맵 불러오기	1	× 3	420
	영역계산(shift 연산)	6		
	영역 내 위치 계산	1		
	자식비트 존재여부 확인 (shift 연산)	**32**		
	누적 값 ADD	1		
	중간합계 ADD	2		
	마스크 초기화	1		
	마스크와 비트맵 비교 (shift) 1 **마스크 shift** 1 ×2 **루프** 1	**96**		
향상된 비트맵 트라이 검색 알고리즘	비트맵 불러오기	1	× 2	90
	영역 계산	6		
	바이트 위치 계산 (shift 연산)	3		
	바이트 영역 계산 (shift 연산)	3		
	바이트 내 위치 계산	4		
	자식비트 존재여부 확인 (shift 연산)	4		
	누적 값 ADD	1		
	마지막 바이트인지 비교 1 카운트 테이블 참조 2 합산 1 ×4 루프 1	20		
	마스크테이블 참조	2		
	바이트 와 마스크의 AND 연산	1		
	카운트 테이블 참조	2		
	ADD 연산	1		

메모리 접근지연에 대하여 알아보자. 우선 각 알고리즘들이 33,858개의 엔트리를 갖는 AADS의 라우팅 테이블로부터 포워딩 테이블을 구축할 때 요구되는 메모리 요구량은 <표 5>와 같다. 표에서 보듯이, 가장 고전적인 이진 트라이의 경우에는 그 크기가 약 2MBytes에 달한다. 이는 본 연구에서 상정한 512KBytes 캐시의 4배에 이른다. 따라서 캐시가 저장할 수 있는 라우팅 정보가 전체의 25% 정도에 불과하다. 더욱이 각 패킷에 대하여 이진 트라이를 검색할 때 데이터 접근에 대한 지역성이 매우 약하다. 이러한 두 가지 이유로 인하여 캐시 접근실패에 따르는 메모리 접근지연이 캐시 접근 성공에 따르는 캐시 접근지연의 148%에 이른다.

반면에 기존 비트맵 알고리즘과 향상된 비트맵 검색 알고리즘은 공히 512Kbytes L2 캐시에 완전히 수용되기에 충분히 적은 양의 메모리를 요구한다. 우리의 알고리즘이 카운트 테이블의 추가에도 불구하고 비트맵 알고리즘보다 적은 양의 메모리를 요구하는 이유는, 성능향상을 위해 추가된 카운트 테이블이 256Bytes에 불과하며 또한 트랜스퍼 테이블에서 중복되는 정보를 제거함으로써 테이블의 크기

를 45% 축소시켰고, 접근빈도수가 낮은 하위의 비트맵을 두 단계로 나누어, 보다 작은 크기의 비트맵으로 구성하였기 때문이다.

이와 같이 캐시보다 작은 용량의 데이터 크기를 갖는 경우에는 일단, 모든 데이터가 캐시에 저장된 이후에는 블록의 교체 없이 지속적으로 사용될 수 있다. 그러나 캐시 접근실패가 관찰되는 이유는 갱신된 포워딩 정보를 접근할 때는 모든 새로운 정보에 대하여 최초의 접근은 캐시 접근실패를 낳는다. 이러한 포워딩 정보의 갱신은 주기적으로 반복되므로 캐시 접근실패에 따르는 메모리 접근지연을 전체 성능에 반영하여야 하는 것이다. 이 뿐만 아니라 캐시 블록 맵핑을 set associative 방식으로 하였으므로 모든 캐시 블록이 채워지기 전에 블록의 교체가 발생할 수 있으며 이로 인한 접근실패에 따르는 메모리 접근지연도 중요한 성능 요소가 된다.

〈표 5〉 각 알고리즘의 메모리 요구량과 그 비율

	이진 트라이	기존 비트맵 트라이	향상된 비트맵 트라이
메모리 요구량	2,001Kbyte	469Kbyte	430Kbyte

마지막으로 <표 2>에서 보듯이 각 알고리즘에 대한 한 패킷 당 포워딩 되어지는 시간을 보여준다. 여기에서 보듯이 향상된 비트맵 트라이 알고리즘은 66ns을 필요로 한다. 앞의 2.1절에서 에서 언급하였던 10기가비트 라우터를 구축하기 위한 패킷당 포워딩 지연시간은 저어도 100ns이하이어야 함을 감안할 때, 본 연구에서 제안하는 알고리즘은 그 조건을 만족하고 있음을 알 수 있다.

7. 결 론

본 논문에서는 향상된 비트맵 트라이를 이용한 프리픽스 검색 알고리즘을 소개하였다. 이 알고리즘은 우선 메모리 요구량을 최소화하기 위하여 라우팅 정보를 500KBytes 이내로 압축하여 일반적인 캐시에 적재될 수 있도록 하였다. 그리고 카운트 테이블을 이용하여 기존 알고리즘의 성능 저하의 가장 큰 요소인 과도한 시프트 연산을 제거하였다. 아울러 포트정보가 저장된 트랜스퍼 테이블로부터 중복된 정보를 제거함으로써 메모리 효율을 향상시켰으며, 주소 검색범위를 다원화하여 데이터 접근 횟수를 최적화하였다.

또한 본 연구에서는 실행-구동 시뮬레이션 기법을 채택하였으며 라우터의 아키텍처를 최대한 정밀하게 모델링하였다. 이 방법을 통하여 검색과정에서 발생하는 명령어 처리지연, 캐시 접근지연 및 메모리 접근지연 등을 구분하여 각각을 CPU 명령어 사이클 단위로 측정할 수 있도록 하였다. 본 연구에서는 최우선적으로 시뮬레이션 결과를 실제 시스템에서 측정된 결과와 비교하여 시뮬레이터 및 시뮬레이션 환경에 대한 검증을 마쳤다. 이와 같이 면밀한 성능진단을 토대로 각 지연요소를 유발하는 원인을 최소화함으로써 주소 검색시간이 기존 알고리즘보다 약 82% 단축된 알고리즘을 개발하였다.

본 연구를 수행하는 과정에서 어려웠던 점은 신뢰할 수 있는 라우팅 테이블과 패킷 트레이스 등 벤치마킹 작업부하를 확보하는 일이었다. 특히 이들을 여러 개 확보하여 다양한 작업부하에 대하여 본 연구를 통하여 개발한 알고리즘을 심도 깊게 검증할 수 있었다면 더욱 신뢰도가 높은 알고리즘을 개발할 수 있었을 것이다.

끝으로, 본 논문에서는 IPv4 환경 아래에서의 라우터의 고속화에 초점을 두었는데, 향후 IPv6환경에서도 적용 할 수 있는 비트맵 트라이 구조에 대한 연구가 진행될 것이다. 또한 카운트 테이블 등을 하드웨어로 구현하여 주소 검색에 필요한 시간을 보다 단축시키는 연구가 진행되고 있다. 아울러, 본 연구를 통하여 구축한 실행-구동 시뮬레이션 환경의 안정도를 높여 이 분야를 연구하는 많은 사람들이 공유할 수 있는 기반을 마련하고자 한다.

참 고 문 헌

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, "Data Structures and Algorithms," Addison-Wesley Pub Co, pp.17-18, pp.78-81, Jan., 1983.

[2] L. Bhuyan and H. Wang, "Execution-Driven Simulation of IP Router Architectures," NCA2001. IEEE International Symposium, pp.8-10, Oct., 2001.

[3] I. L. Chvets and M. H. MacGreaor, "Multi-zone caches for accelerating IP routing table lookups," In Proc. High Performance Switching and Routing, pp.121-126, Jun., 2002.

[4] M. Degermark, et al., "Small Forwarding Tables for Fast Routing Lookups," In Proc. ACM SIGCOMM '97, pp.3-14, Oct., 1997.

[5] W. Eatherton, Z. Dittia, G. Varghese, "Tree Bitmap : Hardware/Software IP Lookups with Incremental Updates," Thesis, Washington "University in St. Louis," pp.7-10, 1998.

[6] S. Karlin, L. Peterson "VERA : An Extensible Router Architecture," IEEE OPENARCH01, pp.3-14, Apr., 2001.

[7] B. Lampson, V. Srinivasan and G. Varghese, "IP Lookups using Multiway and Multicolumn Search," IEEE/ACM Transactions on Networking, Vol.7, No.3, pp.324-334, Jun., 1999.

[8] A. J. McAuley, P. Francis. "Fast routing table lookup using CAMs," In Proceedings of the Conference on Computer Communications, Vol.3, pp.1382-1391, Mar.-Apr., 1993.

[9] D. R. Morrison, "PATRICIA-Practical Algorithm to Retreive Information Coded In Alfanumeric," Journal of the ACM, pp.514-534, Oct., 1968.

[10] A. McAuley, P. Tsuchiya, and D. Wilson, "Fast multilevel hierarchical routing table using contentaddressable memory," U.S. Patent Serial Number, 034444, Dec., 1995.

[11] S. Nilsson, G. Karlsson, "Fast address lookup for Internet routers," In Proc. IEEE Broadband Communications, pp.5-8, Apr., 1998.

[12] S. Oh, J. Ahn, "Bit-Map Trie : A Data Structure for Fast Forwarding Lookups," GlobeCom, pp.1872-1876, 2001.

[13] M. A. Ruiz-Sanchez, E.W. Biersack and Walid Dabbous, "Survey and taxonomy of IP address lookup algorithms," IEEE Network, Vol.15, Issue 2, pp.8-23, Mar.-Apr., 2001.

[14] A. Sharma, "Augmint-A Multiprocessor Simulation Environment for Intel x86 architectures," Technical report, University of Illinois at Urbana-Champaign, Mar., 1996.

[15] S. Venkatachary and G. Varghese, "Faster IP Lookups using Controlled Prefix Expansion," In Proc. of ACM Sigmetrics '98, pp.1-10, June, 1998.

[16] M. Waldvogel, G. Varglese, J Turner and B. Plattner, "Scalable High Speed IP Routing Lookups," In Proc. of ACM SIGCOMM'97, pp.5-9, 1997.

[17] J. Yang, N. Uzun and S. Papavassiliou, "The Architecture Design for a Ten Terabit IP Switch Router," In Proc. IEEE Workshop on High Performance Switching and Routing (HPSR2001), pp.358-362, May, 2001.

[18] IPMA(Internet Performance Measurement and Analysis), http://www.merit.edu/ipma/routing_table/.

이 강 우

e-mail : klee@dgu.ac.kr

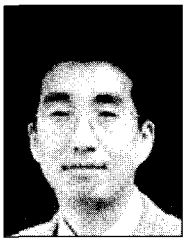
1985년 연세대학교 전자공학과(학사)

1991년 University of Southern California 컴퓨터 공학 석사

1997년 University of Southern California 컴퓨터 공학 박사

1998년~현재 동국대학교 정보통신공학과 조교수

관심분야 : 컴퓨터 구조, 임베디드 시스템



안 종 석

e-mail : jahn@dgu.ac.kr

1983년 서울대학교 전자공학과(학사)

1985년 한국과학기술원 전기 및 전자공학과 석사

1995년 University of Southern California 컴퓨터 공학 박사

1983년~1995년 삼성전자 선임연구원

1996년~현재 동국대학교 컴퓨터공학과 부교수

관심분야 : 실시간 프로토콜, 네트워크 시뮬레이션