Counted Bitmap 기반 초고속 IP 룩업

김재열, 이강우
동국대학교 정보통신공학과
e-mail:{ypicard, klee@dgu.ac.kr}

Fast IP Lookup Based on Counted Bitmap

Jaeyoul Kim, Kangwoo Lee
Department of Information Communications, Dongguk University

요 약

인터넷 회선이 빨라짐에 따라 고속라우터에 대한 요구가 증가하고 있다. 본 논문에서는 고속 라우팅의 핵심인 포워딩 룩업의 고속화를 위하여 비트맵을 이용한 기존의 룩업 알고리즘의 문제점을 실행-구동 시뮬레이션을 통하여 정확히 진단한 후, 카운티드 테이블을 추가하고, 룩업과정에서 사용되는 트랜스퍼 테이블의 중복된 라우팅 정보를 제거함과 아울러 주소 검색범위를 다원화함으로써 성능을 획기적으로 향상시킨 알고리즘을 소개한다. 이 방법은 기존 알고리즘의 룩업시간을 최소 46%, 최대 18%로 단축시키며, 카운티드 테이블을 간단한 하드웨어로 구현한다면 보다 향상된 성능을 기대할 수 있을 것이다.

1. 서론

인터넷 회선이 빨라지면서 고속라우터에 대한 요구가 증가하고 있다. 라우터에서 수신된 패킷의 목적지 주소에 따라 출력포트를 결정하는 룩업은 대규모 정보에 대한 검색으로써 네트워크의 병목이 되므로 라우터 고속화에 대한 연구는 룩업시간 단축에 초점이 맞추어지고 있다.

고속라우터에 관한 연구는 하드웨어, 프로토콜 또는 소프트웨어 연구로 구분되며 소프트웨어 연구에는 두 가지 방법이 있다. 즉, 룩업 알고리즘을 개선하여 명령어 처리지연을 줄이는 방법[3][4][6]과 라우팅 정보를 압축하여 메모리지연을 최소화하는 방법이 있다 [1][2][5][6].

본 논문은 압축을 이용한 방법 중 하나인 비트맵 알고리즘[6]의 문제점을 적시하고, 카운티드 테이블을 추가하고, 룩업에 사용되는 트랜스퍼 테이블의 중복된 라우팅 정보를 제거함과 아울러 주소 검색범위를 다원화함으로써 룩업시간을 기존의 알고리즘들과 비교하여 최소 46%, 최대 18%로 획기적으로 단축시킨 방안을 소개한다.

한편 기존의 연구는 시뮬레이션을 한 경우 시뮬레이션 결과와 실제 성능간의 편차로 인하여 신뢰도가 낮으며, 실제시간을 측정한 경우에는 룩업의 여러 단계 중 병목의 위치와 원인을 정확히 규명하지 못하고 있다. 본 연구에서는 시뮬레이션 기법 중 신뢰도가 가장 높은 실행-구동 시뮬레이션으로 각 룩업 단계에서의 성능을 CPU 명령어 사이클 단위로 측정하였고, 최우선적으로 시뮬레이션 결과를 실제 시스템에서의 결과와 비교하여 결과에 대한 신뢰도를 확보하였으며 이러한 정확한 성능진단을 토대로 병목파악은 물론 그를 제거하기 위한 방안을 도출하였다.

본 논문의 2장은 관련된 연구를 소개하고, 3장에서는 비트맵 알고리즘과 4장에서는 본 논문이 제시하는 카운티드 비트맵 알고리즘을 설명한다. 5장은 시뮬레이션환경 및 방법을 설명하고 실험결과 및 결과에 대한 분석은 6장에 제시하였다. 7장의 결론을 끝으로 본 논문을 마친다.

2. 관련 연구

소프트웨어에 기반한 라우팅의 고속화에는 두 가지 접근 방법이 있다. 즉, 알고리즘을 개선하여 명령어 처리지연 및 데이터 접근지연을 최소화하거나 라우팅 정보를 압축하여 메모리지연을 최소화하는 방법이 있다.

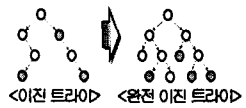
룩업 알고리즘에 대한 연구 중 [4]는 빈번한 메모리 접근으로 인하여 패킷 처리시간이 매우 길다. [3]에서는 프리픽스의 길이를 모두 같도록 하여 최대 6중 룩업을 실시한다. 또한 [8]에서는 같은 길이를 갖는 프리픽스별로 여러 개의 트라이를 구축하였으며, [9]는 프리픽스 길이별로 여러 개의 테이블을 두어 우선 프리픽스 길이에 대한 룩업을, 그리고 해당 테이블에서의 검색을 시행한다.

라우팅 정보압축에 대해 [5]에서는 두 단계에 걸쳐 트라이 레벨을 압축한 LC-트라이를 제안하였으며 [1]은 트라이를 두 개로 분리하고 중복되는 비트열을 공통으로 표현하여 크기를 줄였다. [2]에서는 포워딩 테이블을 캐시에 저장할 수 있을 만큼 압축한 비트맵이라는 구조를 처음으로 채택하였으며 [6]에서 이를 발전시켜 비트맵 트라이를 이용한 룩업 알고리즘을 제안하였다.

3. 비트맵 룩업 알고리즘

3.1 비트맵 트라이

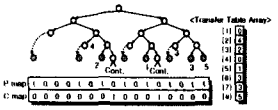
이진트라이로부터 모든 노드가 자식노드를 0개 또는 2개를 갖도록 완전이진트라이를 만든다. 즉, [그림 1]과 같이 라우팅 정보를 갖는 원소노드의 자식이 0개 또는 2개라면 그대로 두고 자식이 하나인 경우 부족한 자식을 추가하며, 자식노드(들)가 라우팅 정보를 갖지 않는 비원소



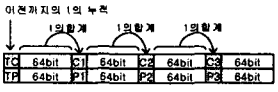
[그림 1] 완전이진 트라이

노드(들)라면 라우팅 정보를 자식(들)에게 상속해주고 자신은 비원소노드로 남는다[6].

비트맵을 생성하는 과정은 다음과 같다. 완전이진 트라이에 대하여 중위순회를 실행하여 원소노드가 나타나면 비트열에 '1'을 추가하고, 또한 원소노드에 해당하는 라우팅 정보를 트랜스퍼테이블이라는 배열에 저장한다. 최하위 노드의 높이를 0이라 할 때, 원소노드의 높이가 h인 경우 '1' 뒤에 2^h-1개의 '0'을 추가한다. 이를 P-map이라 하며 완전이진 트라이를 8레벨 단위로 나누어 각 서브트라이별로 P-map을 구축한다. 또한 자식을 가진 비원소노드에는 '1', 아니면 '0'을 배정한 또 다른 비트맵을 만들어 C-map이라 부르고 이는 하위 P-map 인덱스를 구하는데 사용한다. 한편 보다 빠른 룩업을 위하여, 각 비트맵에는 [그림 3]과 같이 그 비트맵에 도달하기까지의 1의 개수의 누적과, 64비트씩 4구간으로 나누어 각 구간의 1의 값을 합산한 구역합계를 함께 저장한다.



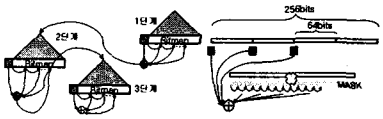
[그림 2] P-map과 C-map의 예시



[그림 3] 비트맵 구조

3.2 비트맵 트라이 룩업 알고리즘

패킷의 목적지 주소의 첫 8비트의 값이 x일 때, 최상위 C-map의 x번째 비트의 값이 1이면, 이 C-map의 처음부터 x번째 비트까지의 1의 개수가 다음에 검사해야할 하위 C-map의 인덱스가 된다. 선택된 하위 C-map에서 입력 패킷의 다음 8비트 값에 대하여 같은 방식으로 검사가 이루어지며, 해당 비트가 0인 경우, 최상위 P-map부터 현재 P-map의 해당 비트까지의 1의 개수가 트랜스퍼 테이블의 인덱스 값으로 사용되어, 트랜스퍼 테이블로부터 원하는 프리픽스 정보를 얻는다. [그림 4]는 비트맵 룩업과정을 보여주며, 보다 자세한 룩업과정은 [6]에서 볼 수 있다.



[그림 4] 비트맵 트라이의 룩업 과정

3.3 비트맵 트라이 룩업 알고리즘의 문제점

비트맵 알고리즘의 첫 번째 단점은, 두개의 비트맵에서 1의 개수를 세기 위한 시프트 연산이 많아 룩업시간이 길어진다는 것이다. ([표 2]에 표시) 이는 [표 1]에서 보듯이 통상적인 이진 트라이에 비해 메모리 요구 량은 적지만 시프트 연산으로 인한 명령어 처리지연이 급증함으로써 룩업 시간은 오히려 증가하는 결과를 초래한다.

	메모리요구	명령어 수	캐시접근 수	메모리접근 수	계
이진검색	2,001KB	146,637,431	25,350,052	36,995,552	208,983,035
비트맵	469KB	472,048,634	11,819,214	1,637,578	485,505,426

[표 1] 이진검색/비트맵 룩업 실행 시간 구성

둘째로, [6]에서 사용된 트랜스퍼 테이블에는 동일한 정보가 중복되어 저장되어 있다. 이로 인하여 메모리 낭비되며 결국은 메모리 접근지연의 증가를 초래한다.

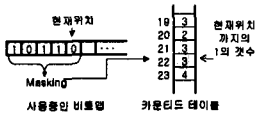
끝으로, 비트맵의 라우팅 정보는 그 위치에 따라 접근빈도에 상당한 편차를 보인다. 즉, 상위레벨의 비트맵들은 거의 매 패킷마다 사용되므로 빈번하게 접근되지만, 하위레벨의 비트맵들은 사용 빈도가 낮다. 이와 같은 현상을 고려하지 않고 각 단계별 비트맵의 크기를 동일하게 함으로 해서 향상된 성능을 얻을 수 있는 요소를 간과하였다.

4. 카운티드 비트맵 룩업 알고리즘

본 연구에서는 3.3절의 기존 비트맵 알고리즘의 문제들을 제거하여 룩업 성능이 향상된 알고리즘을 개발하였다.

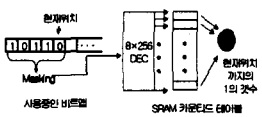
4.1 카운티드 테이블을 이용한 시프트 연산의 감축

룩업과정 중 P-map과 C-map에서 얻고자 하는 것은 해당 위치까지의 1의 총 개수이다. 이를 위하여 기존의 알고리즘에서는 많은 시프트 연산을 수행하였다. 본 연구에서는 시프트연산을 감소시키기 위해 8비트 이진수가 표현하는 256가지 경우마다의 1의 개수를 「카운티드 테이블」에 저장하였다. 이는 [그림 5]와 같이 룩업에 사용되어지고 있는 비트맵의 현재까지의 비트열을 마스킹한 후, 이 이진수 값에 해당하는 카운티드 테이블로부터 1의 개수를 빠르게 얻을 수 있다.



[그림 5] 카운티드 테이블의 사용

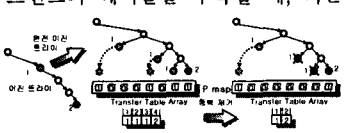
한편, 이 알고리즘에서는 하나의 카운티드 테이블만 필요하므로 추가되는 메모리 공간은 256Byte에 불과하다. 이 테이블은 그 크기가 매우 작으므로 [그림 6]과 같이 하드웨어로 구현하면 보다 빠른 룩업을 달성할 수 있다.



[그림 6] 하드웨어 카운티드테이블

4.2 트랜스퍼 테이블의 축소

기존의 비트맵 알고리즘은 완전이진트라이를 구축할 때 [그림 7]과 같이 동일한 라우팅 정보를 가진 원소노드가 여러 개 생성된다. 이에 따라 트랜스퍼 테이블에는 불필요하게 중복된 정보가 저장된다. 이러한 중복된 정보를 제거하기 위하여, 트라이에 대하여 중위순회를 실행하며 트랜스퍼 테이블을 구축할 때, 최근의 라우팅정보와 동일한 라우팅정보를 갖는 원소노드를 발견하면 이의 정보는 테이블에 저장하지 않으며, P-map에도



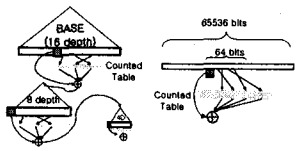
[그림 5] 트랜스퍼 테이블의 축약

1을 발생시키지 않는다. 이 방식으로 중복된 모든 정보를 제거하지는 못하였지만, 테이블의 크기를 50%이상 감소시키는 효과를 보았다. 이러한 메모리 요구량의 감소에 의해 메모리접근지연도 감소시키는 효과를 얻었다.

4.3 IP 주소 룩업범위의 다원화

트라이의 상위 레벨은 접근빈도가 높으므로 큰 깊이의 비트맵을, 최하위의 비트맵들은 접근빈도가 매우 낮으므로 작은 깊이의 비트맵을 구성할 필요가 있다. 또, 중요한 사실은 통계적으로 가장 사용 빈도가 높은 원소노드는 16~24레벨에 위치하고 있다는 것이다.

본 연구에서는 비트맵을 16-8-4-4의 깊이로 구분하여 구성하였다. 한편 16레벨로 구성된 비트맵을 64비트 영역으로 나누어 1의 개수에 대한 중간합계를 저장하면, 룩업시 최대 1024번의 ADD연산이 발생한다. 본 연구에서는 중간합계대신 누적합계를 저장하여, 한번의 데이터 접근으로 1의 개수를 얻도록 하였다.



[그림 8] 룩업범위의 다원화

4.4 카운티드 테이블을 이용한 룩업 알고리즘

```
1:  Let routing entry pointer = default routing entry;
2:  for (addr = IP_Address[0-15:16-23:24-27:27-31])
3:      get BM_node = BM(bm_idx);
4:      get position_idx = addr mod 64;
5:      get byte_idx = addr / 8;
6:      get region_idx = byte_idx / 8;
7:      get bit_idx = addr mod 8;
8:      if (BM_node's c-bitmap's byte(byte_idx) & (0x80 >> bit_idx))
9:          bm_idx = tc + c[region_idx];
10:         for (i = region_idx to byte_idx-1) bm_idx += CT[c-map(i)];
11:         bm_idx += CT[ c-map(byte_idx) & MT[bit_idx] ];
12:     else
13:         transfer_idx = tp + p[region_idx];
14:         for (i = region_idx to byte_idx-1) transfer_idx += CT[p-map(i)];
15:         transfer_idx += CT[ p-map(byte_idx) & MT[bit_idx] ];
16:         return transfer_table[transfer_idx];
```

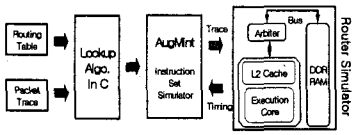
BM[]: 비트맵 CT: 카운티드 테이블 MT: 마스크 테이블

[그림 9] 카운티드 비트맵 기반 룩업 알고리즘

본 연구에서 제안하는 알고리즘을 간단히 설명하면 다음과 같다. 패킷의 목적지 주소의 첫 16비트 값에 해당하는 최상위 C-map의 비트 값이 1인가 검사하여 자식노드가 있는지 확인한다. 자식노드가 있으면 카운티드 테이블을 이용하여 C-map에서 현재위치까지의 1의 개수를 이용하여 다음 레벨 C-map의 인덱스를 구한다. 목적지 주소의 다음 8비트 값과 다음 단계 C-map에 대하여 최하위 C-map까지 반복적으로 검사한다. C-map의 해당위치 비트 값이 0이면, 그 위치에서의 P-map과 카운티드 테이블을 이용하여 트랜스퍼테이블 인덱스를 구한다. 끝으로, 트랜스퍼 테이블로부터 출력포트의 정보를 구한다.

5. 시뮬레이션 환경

본 연구에서는 [그림 10]과 같이 AugMint[7]에 기반한 실행-구동 시뮬레이션 환경을 구축하였다. C언어로 구현된 룩업 알고리즘을 AugMint가 가상적으로 실행하고 그 트레이스를 라우터 시뮬레이터에 제공하며, 프로세서·캐시·메모리 및 버스에서의 동작을 시뮬레이트하여 실행에 따르는 시간 정보를 피드백 하여 룩업의 실행시간을 조정한다.



[그림 10] 시뮬레이션 환경

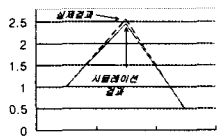
33,858개의 엔트리를 갖는 AADS의 테이블을 사용하였고, 이 라우터를 통과한 약 120만 패킷의 목적지 주소 트레이스를 사용하였다.

여기서 고려한 시스템은 인텔의 펜티엄 4이며, L2 캐시는 Write-back, 8-way set associative, 64byte cache line으로 512KByte이다. 메모리는 DDR 2100 SDRAM 을 기준으로 133MHz PCI 버스의 두 배의 접근속도를 가지게 하였다. 운영체제는 커널버전 2.4.18 의 레드햇-리눅스를, 컴파일러는 GNU gcc v3.2를 -O1 옵션으로 하였다.

6. 시뮬레이션 결과

우선 시뮬레이션 결과의 신뢰성을 확보하기 위하여, 세 가지 알고리즘에 대하여 시뮬레이션을 통한 CPU 명령어 사이클 수와 실제속도를 측정하였고, 각 값을 이진검색 알고리즘의 값으로 정규화 하여 [그림 11]에 제시하였다. 본 연구의 목적이 각 알고리즘들의 상대적인 성능 비교에 있음을 감안할 때, 각 알고리즘의 정규화된 실행 시간 비율을 통하여 본 연구에서 수행한 시뮬레이션 결과에 대한 신뢰성을 부여할 수 있음을 알 수 있다.

	이진검색	비트맵	카운티드 비트맵
시뮬레이션(cycle) / normalization	181,574,061 / 1	461,088,286 / 2.539	83,673,152 / 0.461
실제측정(ms) / normalization	173.828 / 1	426.484 / 2.453	83.984 / 0.483



[그림 11] 알고리즘간 룩업시간 비교

각 알고리즘의 룩업시간을 비교하면 [그림 12]와 같다. 룩업시간은 명령어 처리지연과 캐시 접근지연, 그리고 메모리 접근지연(캐시미스 페널티)으로 나누어진다.

	이진 검색	비트맵	카운티드 비트맵
명령어 지연	119,228,457	447,631,494	70,998,239
캐시접근 지연	25,187,076	11,812,000	11,139,333
캐시미스 Penalty	37,158,528	1,644,792	1,535,580
합계	181,574,061	461,088,286	83,673,152

[그림 12] 각 알고리즘의 룩업시간 비교

[그림 12]에서 보면 비트맵 알고리즘의 명령어 처리지연은 이진검색의 명령어 지연 보다 3배 가량 많다. 이로 인하여 라우팅 정보를 압축함으로써 이진검색보다 절대적으로 적은 메모리 접근지연을 달성했음에도 불구하고 시프트연산으로 인하여 오히려 성능이 저하되는 결과를 초래하였다. 한편, 본 연구에서 제시한 카운티드 비트맵을

이용한 알고리즘은 룩업시간 측면에서 본다면 이진검색 알고리즘과 비트맵 알고리즘에 비교하여 각각 46%, 18%에 불과할 만큼 우수한 룩업성능을 보여준다.

구분	지연요소	지연	평균단계	합계
이진트라이	현노드 인기 확인 비트추출 비트비교 (1 or 0) 포인터 얻어오기 노드 불러오기	1 1 1 1 1	×24	120
비트맵 트라이 룩업 알고리즘	비트맵 불러오기 영역계산(shift OP) 영역 내 위치 계산 자식비트 존재여부 확인 (shift OP) 누적값 더하기 중간합계 더하기 마스크 초기화 마스크와 비트맵 비교(shift) 마스크 쉬프트 루프	1 6 1 32 1 2 1 1 1 1 ×32	×3	420
카운티드 비트맵 트라이 룩업 알고리즘	비트맵 불러오기 영역 계산 바이트 위치 계산 (shift OP) 바이트 영역 계산 (shift OP) 바이트 내 위치 계산 자식비트 존재여부 확인 (shift OP) 누적값 더하기 마지막 바이트인지 비교 카운티드테이블 참조 합산 루프 마스크테이블 참조 바이트 와 마스크의 and 연산 카운티드테이블 참조 합산	1 6 3 3 4 4 1 1 2 ×4 1 20 2 1 2 1	×2	90

[표 2] 검색 알고리즘의 처리 지연 요소 비교 (명령어 수)

한편, 비트맵 룩업 알고리즘과 우리가 제안하는 카운티드 비트맵 알고리즘의 캐시접근 수는 서로 비슷하여 성능의 차이에 크게 영향을 미치지 않는다. 그러나 이진검색 알고리즘의 캐시접근 수에 비교하면 약 46%~48% 정도에 불과하여 성능 향상에 도움을 준다.

끝으로 메모리 접근 지연요소에 대하여 알아본다. 우선 각 알고리즘들이 33,858개의 엔트리를 갖는 AADS의 라우팅 테이블로부터 포워딩 테이블을 구축할 때 요구되는 메모리 요구량은 [표 3]과 같다. 표에서 보듯이, 가장 고전적인 이진 트라이의 경우에는 포워딩 테이블의 크기가 약 2Mbyte에 달하며 이는 본 연구에서 상정한 512KByte L2 캐시의 4배에 이른다. 반면에 비트맵 알고리즘과 우리가 제안하는 카운티드 비트맵 알고리즘은 공히 512KByte L2 캐시에 완전히 수용되기에 충분히 적은 양의 메모리를 요구한다. 우리의 알고리즘이 카운티드 테이블의 추가에도 불구하고 비트맵 알고리즘보다 적은 양의 메모리를 요구하는 이유는 카운티드 테이블이 256Byte에 불과하며 또한 4.2절에서와 같이 트랜스퍼 테이블에서 중복되는 정보를 많이 제거하여 테이블의 크기를 축소시켰기 때문이다.

	이진 트라이	비트맵 트라이	카운티드 비트맵 트라이
메모리 요구량	2,001KByte	469KByte	430KByte
normalization	1	0.23	0.21

[표 3] 각 알고리즘의 메모리 요구량과 그 비율

[표 4]는 각 알고리즘에서 발생하는 캐시미스 수를 보여준다. 비트맵 알고리즘과 카운티드 비트맵 알고리즘은 데이터의 크기가 캐시보다 작아서 초기(cold)캐시미스만이 발생하며, 그 수가 매우 작아서 전체성능에 미치는 영향은 미미하다. 그러나 이진검색 알고리즘의 경우 데이터가 커서 다수의 교체(replacement)캐시미스가 발생하며 [표 3]에 의하면 전체 성능의 31%의 지연요소가 된다.

	이진 트라이	비트맵 트라이	카운티드 비트맵 트라이
캐시 미스 수	162,976	7,214	6,735
normalization	1	0.044	0.041

[표 4]각 알고리즘의 캐시 미스 수

7. 결론

고속 라우팅 룩업을 위하여 본 연구에서는 압축을 이용한 방법 중 하나인 비트맵 알고리즘의 문제점을 지적하고 이를 극복하기 위한 방안으로 카운티드 테이블을 추가하고, 룩업에 사용되는 트랜스퍼 테이블의 중복된 라우팅 정보를 제거함과 아울러 주소 검색범위를 다원화하는 등 다각적인 측면에서 개선된 알고리즘을 제안하였다.

또한 본 연구를 수행함에 있어서 면밀한 아키텍처 모델링과 실행-구동 시뮬레이션을 통하여 각 룩업 단계에서의 성능을 CPU 명령어 사이클 단위로 측정하였다. 모든 실험은 시뮬레이션 결과를 실제 시스템에서의 결과를 비교하여 결과에 대한 신뢰도를 확보한 후 수행하였으며 이와 같은 정확한 성능진단을 토대로 병목 파악은 물론 그를 제거하기 위한 방안을 도출하였다.

그 결과 우리는 룩업시간을 기존의 알고리즘들과 비교하여 최소 35%, 최대 17%로 획기적으로 단축시킨 알고리즘을 개발할 수 있었다. 그리고 본 연구에서 제안한 카운티드 테이블 등을 간단한 하드웨어로 구현한다면 보다 향상된 성능을 기대할 수 있을 것이다.

참고문헌

[1] M. Degermark, *et al*, "Small Forwarding Tables for Fast Routing Lookups," In Proceedings of ACM SIGCOMM'97, pp.6-14, Oct. 1997

[2] 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.

[3] B.Lampson, V.Srinivasan and G.Varghese,"IP Lookups using Multiway and Multicolumn Search,"In Proc. of ACM Sigmetrics'98, pp.12-15, June 1998

[4] Donald R. Morrison, "PATRICIA - Practical Algorithm to Retreive Information Coded In Alfanumeric,"journal of the ACM, pp.514-534, October 1968

[5] Stenfan Nilsson, Gunnar Karlsson, "Fast address lookup for Internet routers,"In proceedings of IEEE Broadband Communications'98, pp.5-8, April 1998

[6] Seunghyun Oh, Jongsuk Ahn, "Bit-Map Trie : A Data Structure for Fast Forwarding Lookups,"Technical Program of Internet Performance Symposium 2001, pp.3-7, 2001

[7] Arun Sharma, "Augmint - A Multiprocessor Simulation Environment for Intel x86 architectures," CDRD Technical Report, pp.1-3, March 1996

[8] S.Venkatachary and G.Varghese,"Faster IP Lookups using Controlled Prefix Expansion,"In Proc. of ACM Sigmetrics'98, pp.7-13, June 1998

[9] Marcel Waldvogel, George Varglese, Jon Turner and Bernhard Plattner, "Scalable High Speed IP Routing Lookups," In Proc. of ACM SIGCOMM'97, pp.5-9, 1997