# 패킷 분류를 위한 스마트 셋-프루닝 트라이

준회원 민 세 원*°, 이 나 라*, 종신회원 임 혜 숙*

# A Smart Set-Pruning Trie for Packet Classification

Seh-won Min*°, Nara Lee* *Associate Members*, Hyesook Lim* *Lifelong Member*

요 약

패킷분류는 라우터의 가장 기본적이면서도 중요한 기능 중의 하나이며, 실시간 전송을 요구하는 새로운 인터넷 응용 프로그램의 등장과 더불어 그 중요성이 더욱 커지고 있다. 패킷분류는 입력 패킷에 대하여 선속도로 이루어져야 하며, 여러 헤더 필드에 대해 다차원 검색을 수행해야 하기 때문에 라우터 설계의 어려운 문제 중에 하나이다. 고속의 패킷분류를 제공하기 위한 다양한 패킷분류 알고리즘이 제안되어 왔으며, 그 중 계층적 접근 방식을 사용한 알고리즘은 하나의 필드에 대하여 검색이 수행될 때마다 많은 검색 영역이 제거되기 때문에 효율적이다. 그러나 계층적 구조는 역추적이라는 문제를 내재하고 있으며, 이를 해결하기 위해 사용되는 셋-프루닝 트라이나 그리드-오브-트라이는 지나치게 많은 노드 복사를 야기하거나, 선-계산이라는 복잡한 과정을 요구한다. 본 논문에서는 셋-프루닝 하위 트라이의 간단한 합병을 통하여 복사되는 노드의 개수를 줄일 수 있는 스마트 셋-프루닝 구조를 제안한다. 시뮬레이션 결과 제안된 구조는 셋-프루닝 트라이와 비교하여 복사되는 노드 수 및 룰 수가 2-8% 줄어듦을 확인하였다.

**Key Words** : Smart Set-Pruning, Packet Classification, Hierarchical Trie, Set-Pruning Trie, Scalability

ABSTRACT

Packet classification is one of the basic and important functions of the Internet routers, and it became more important along with new emerging application programs requiring real-time transmission. Since packet classification should be accomplished in line-speed on each incoming input packet for multiple header fields, it becomes one of the challenges in designing Internet routers. Various packet classification algorithms have been proposed to provide the high-speed packet classification. Hierarchical approach achieves effective packet classification performance by significantly narrowing down the search space whenever a field lookup is completed. However, hierarchical approach involves back-tracking problem. In order to solve the problem, set-pruning trie and grid-of-trie algorithms are proposed. However, the algorithm either causes excessive node duplication or heavy pre-computation. In this paper, we propose a smart set-pruning trie which reduces the number of node duplication in the set-pruning trie by the simple merging of the lower-level tries. Simulation result shows that the proposed trie has the reduced number of copied nodes by 2-8% compared with the set-pruning trie.

## I. 서 론

기존의 인터넷 라우터가 제공하는 베스트 에포트 (best-effort) 서비스는 최근에 등장한 실시간 전송을 요구하는 새로운 응용프로그램의 서비스에는 적합하지 않은데, 이는 입력 패킷의 구분 없이 서비스의

* 이화여자대학교 전자공학과(ideakrys@naver.com, detler@hanmail.net, hlim@ewha.ac.kr), (°: 교신저자)
논문번호 : KICS2010-11-562, 접수일자 : 2010년 11월 28일, 최종논문접수일자 : 2011년 11월 4일

대역폭이나 전송 시간을 동일하게 처리하기 때문이다. 그에 따라 입력 패킷의 다양한 플로우에 따른 품질 보증(quality of service, QoS)을 제공하는 정책 기반 서비스가 중요시 되고 있으며, 이 때 필수적인 요소가 패킷분류이다. 패킷분류란 라우터에 입력된 패킷을 미리 정의된 룰 셋에 따라 입력된 패킷이 속하는 클래스로 분류하는 것으로, 그 클래스에 속하는 정책에 따라 라우터의 서비스를 제공받을 수 있도록 하여 품질 보증을 가능하게 한다[1].

패킷분류의 어려움에는 여러 가지 요인이 있다. 첫째로, 패킷분류를 위한 클래스는 패킷의 여러 헤더 필드에 속하는 룰들에 의하여 정의되기 때문에, 패킷이 입력되면 여러 개의 헤더 필드에 대해 다차원 검색을 동시에 수행하여야 한다는 점이다[2]. 각각의 헤더 필드에 대하여 일치 여부를 판단하기 위해서 완전 일치(exact match), 영역 일치(range match), 프리픽스 일치(prefix match) 중 적합한 방법에 의하여 일치 여부를 판단하여야 할 뿐만 아니라 일치 가능한 여러 개의 룰 중에서 가장 높은 우선순위를 갖는 룰을 검색하여 그에 따른 정책으로 패킷을 처리해야 하는 과정이 부가적으로 이루어져야 한다. 또한 인터넷 라우터로 입력되는 패킷은 초당 최대 수 천만 개가 될 수 있고, 들어오는 모든 입력 패킷에 대하여 선속도(wire-speed) 프로세싱을 수행하지 않으면 전체 네트워크의 성능을 저하시키는 병목점이 될 수 있기 때문에 빠른 속도로 처리해야 한다는 점에서 어려움이 있다[3]. 이를 해결하기 위해서 라우터 설계에 있어 빠른 패킷분류를 제공하기 위한 효율적인 데이터 구조 및 알고리즘[4]에 대한 연구가 활발히 진행되고 있으며, 라우터의 패킷분류 성능은 각각의 입력 패킷으로부터 적합한 클래스를 검색하는 과정에서 요구되는 평균 메모리 접근 횟수로서 평가된다[5].

패킷분류에 관련되어 제안된 알고리즘 중에서 특히 패킷분류에 사용되는 5 개의 헤더 필드 중 근원지와 목적지 IP 주소를 이용한 계층적 구조 기반의 다양한 알고리즘이 제안되었는데[6], 이는 근원지와 목적지 IP 주소 조합이 가지는 다양성과 그에 따른 효율성에 기반을 둔다[7]. 그러나 기존의 계층적 트라이는 역추적(back-tracking)이 요구되기 때문에 메모리 접근 횟수가 증가하여 개선의 여지가 있다. 셋-프루닝 트라이는 계층적 트라이의 역추적을 제거하여 검색 속도를 증가시킨다는 장점이 있지만 룰 및 노드의 복사로 인한 메모리 증가의 문제가 있다. 본 논문에서는 스마트 셋-프루닝(set-pruning) 기법을 통하여 룰 및 노드의 복사를 줄이면서도 역추적을 제거함으로써 검색 성능을 높이는 스마트 셋-프루닝 패킷분류 구조를 제안하고자 한다.

## Ⅱ. 본 론

### 2.1 패킷분류 문제의 정의(Problem Statement)

라우터가 갖는 분류 테이블에는 $N$개의 룰이 저장되어 있으며, 각각의 룰은 $R=\{R_i|i=0,1,...,N\}$로 표현되어 패킷분류에 사용된다. 분류 테이블 내의 모든 룰들은 우선순위에 따라 순서대로 정렬되어 있으며, 표 1은 실제 인터넷 라우터의 룰 특성을 반영하는 클래스벤치(class-bench)로부터 21개의 룰들을 선택하여 구성한 분류기(classifier)이다. 이 예시를 통해 기존의 패킷분류 구조 및 제안하는 구조를 설명하고자 한다. 이 때 룰 번호가 작을수록 더 높은 우선순위 (priority)를 가짐을 가정하였다[5],[6].

각각의 룰 $R_i$은 $R_i=\{F_i^j|j=1,...,5\}$로 표현되며, 근원지 프리픽스, 목적지 프리픽스, 근원지 포트 번호, 목적지 포트 번호, 프로토콜 타입의 5 개의 필드들의 특정 값들로 이루어져 있다. 이 때 $F_i^j$는 룰 $R_i$의 $j$번째 필드를 나타낸다. 여러 개의 다양한 필드로 구성되어 있는 입력 패킷의 헤더 중에서 패킷분류와 관계있는 5개의 필드를 추출하여 입력 헤

표 1. 분류기 예시
Table 1. An example of classifier set

| Rule | Src Prefix | Dest Prefix | Src Port | Dest Port | Protocol |
|---|---|---|---|---|---|
| $R_0$ | 0* | 101* | 0, 0 | 0, 0 | 6 |
| $R_1$ | 010* | 01* | 0, 0 | 0, 0 | 17 |
| $R_2$ | 0111* | 01* | 0, 0 | 0, 0 | 6 |
| $R_3$ | 111* | 1* | 53, 53 | 443, 443 | 6 |
| $R_4$ | * | 1* | 0, 0 | 0, 0 | 4 |
| $R_5$ | 0000* | 1* | 53, 53 | 443, 443 | 6 |
| $R_6$ | * | 1* | 67, 67 | 443, 443 | 6 |
| $R_7$ | 010* | 1* | 53, 53 | 443, 443 | 6 |
| $R_8$ | 0000* | 1* | 1024, 65535 | 2788, 2788 | 17 |
| $R_9$ | 0000* | 1* | 1024, 65535 | 25, 25 | 6 |
| $R_{10}$ | 0* | 01* | 53, 53 | 443, 443 | 17 |
| $R_{11}$ | 010* | 01* | 67, 67 | 443, 443 | 6 |
| $R_{12}$ | 011* | 01* | 67, 67 | 443, 443 | 6 |
| $R_{13}$ | 011* | 01* | 1024, 65535 | 2788, 2788 | 4 |
| $R_{14}$ | 011* | 010* | 0, 0 | 0, 0 | 6 |
| $R_{15}$ | 0111* | 01* | 1024, 65535 | 25, 25 | 17 |
| $R_{16}$ | 1100* | * | 0, 0 | 0, 0 | 6 |
| $R_{17}$ | 11* | * | 0, 0 | 0, 0 | 6 |
| $R_{18}$ | 111* | 0* | 53, 53 | 443, 443 | 4 |
| $R_{19}$ | 010* | * | 1024, 65535 | 25, 25 | 6 |
| $R_{20}$ | 1100* | 11* | 53, 53 | 443, 443 | 6 |

더로써 $H=(H^1, H^2,...,H^5)$와 같이 표현하며 검색의 주체가 된다. 특정 룰 $R_i=\{F_i^j | j=1,...,5\}$가 갖는 5개의 필드 값에 대해 입력 패킷의 모든 헤더가 $H^j \in R_i^j$의 관계를 갖게 되면 입력 패킷 $H$와 룰 $R_i$가 일치하였다고 정의한다.

각각의 필드에 대하여 일치 여부를 판단하는 방법은 상이하다. 근원지 및 목적지 IP 주소 필드에 대해서는 프리픽스 일치가 수행되어야 하고, 근원지 및 목적지 포트 번호 필드에 대해서는 영역 일치가 수행되어야 하며, 프로토콜 타입 필드에 대해서는 완전일치가 수행되어야 한다. 프리픽스 일치란 입력 패킷의 해당 필드로부터 주어진 프리픽스 길이만을 비교하는 방법을 말하며, 영역일치란 입력 패킷의 해당 필드 값이 특정 룰의 해당 필드의 영역에 포함되는지 여부를 판단하는 방법을 말하고, 완전 일치란 입력 패킷과 특정 룰에 대해서 해당 필드의 값이 정확히 일치하는지 여부를 판단하는 방법을 말한다. 입력 패킷이 하나 이상의 룰과 일치하는 경우 우선순위를 고려하여 최종 결과를 결정하며, 해당 룰의 정책(policy)에 따라 패킷을 처리한다.

예를 들어 입력 패킷 $H$ (0110, 0100, 67, 443, 6)는 $R_{12}$와 일치하며 $R_{12}$보다 더 낮은 인덱스 값을 가지는 일치하는 룰이 존재하지 않으므로 $R_{12}$가 패킷분류의 최종 결과가 된다.

### 2.2 계층 트라이(Hierarchical Trie)

계층 트라이[1]는 룰을 구성하는 5개의 필드 중 근원지와 목적지의 프리픽스 필드인 $F_1$ 필드와 $F_2$ 필드에 대하여 각각 트라이를 구성하고, 상위 트라이인 근원지 주소 트라이에서 하위 트라이인 목적지 주소 트라이로 계층적으로 연결한 트라이이다. 계층 트라이는 트라이에 기초한 구조 중에서도 가장 기본적인 구조이며, 들어오는 입력 패킷에 대해 일치 여부를 판단해야 하는 모든 필드가 프리픽스로 이루어졌다고 가정하기 때문에 프리픽스가 아닌 영역으로 표현되는 포트 번호 필드들에 대해서는 하나 혹은 그 이상의 프리픽스로 변환하여 저장해야 한다.

각 트라이 구조는 프리픽스의 가장 왼쪽에 있는 비트인 *MSB(most significant bit)*부터 한 레벨씩 진행하여 *LSB(least significant bit)*에 도달하면 해당 위치에 프리픽스 노드를 저장하며, 0이면 현재 노드의 왼쪽 노드로, 1이면 오른쪽 노드로 진행한다. 하위 트라이는 첫 번째 필드의 프리픽스 값이 같은 룰들로만 이루어진 트라이로써, 상위 트라이의 프리픽스 노드는 하위 트라이의 포인터 값을 저장하고 있으며 하위 트라이의 노드는 나머지 3개의 필드 값을 모두 저장하고 있다. 하위 트라이에서 하나의 노드에 여러 개의 룰이 존재하는 경우, 즉 처음 두 필드 값이 같은 룰이 여러 개 존재하는 경우, 우선순위대로 연결된 링크드 리스트(linked list)를 통해 나머지 세 개의 필드에 대하여 다른 필드 조합을 갖는 룰의 존재 여부를 나타내는 방식으로 구성한다. 그림 1에서는 표 1의 분류테이블을 바탕으로 근원지 프리픽스와 목적지 프리픽스 필드를 이용한 계층적 트라이를 보였으며, 프리픽스 노드는 검은색, 빈 노드는 흰 색으로 표현하였다. 각 계층 트라

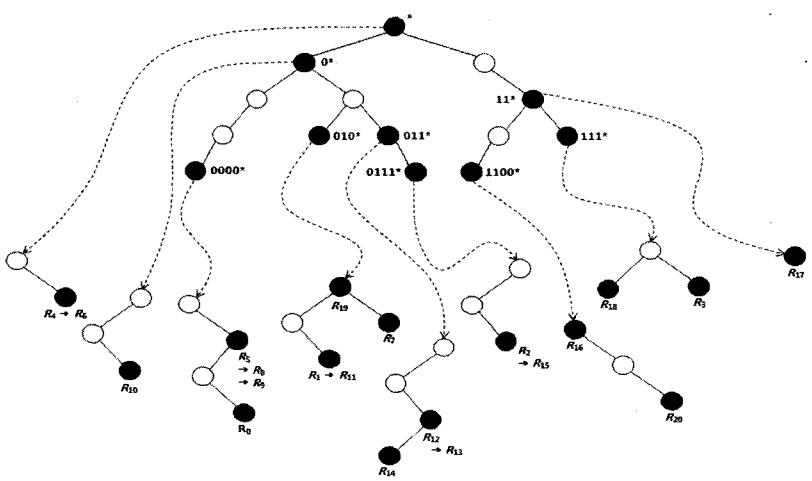



그림 1. 계층 트라이
Fig. 1. Hierarchical Trie

이의 루트 노드부터 잎 노드(leaf node)까지 입력 패킷의 MSB부터 한 비트씩 비교하여 그 값이 0또는 1일 때 각각 왼쪽 또는 오른쪽 포인터를 따라가는 방식으로 검색이 진행되며, $F_1$ 필드에 해당하는 상위 트라이에서 일치하는 모든 노드에 대하여 그 노드에 저장된 포인터를 통해 계층적으로 연결된 하위 필드 트라이로 검색을 확장하며 순차적으로 이루어진다.

각각의 두 필드 값에 대해 모두 일치하면 해당 노드의 나머지 세 개의 필드 값을 비교하여 일치하면 해당 룰을 현재까지의 *BMR(best matching rule)*로 저장한다. 나머지 세 개의 필드 값과 일치하지 않으면 링크드 리스트 포인터의 존재 유무에 따라 나머지 값을 비교하여 일치 여부를 판단하고, 해당 트라이에서의 검색이 종료되면 상위 트라이로 검색 영역을 돌려주는 역추적을 해야 한다. 이는 일치 가능한 룰 중에서도 가장 높은 우선순위를 갖는 룰을 검색하여야 하기 때문이며, 상위 트라이에서 잎 노드를 만날 때까지 반복적으로 진행된다.

계층적 트라이 구조에서 분류기의 룰의 개수를 $N$, 프리픽스의 최대 길이를 $W$라 할 때, 메모리 요구량의 복잡도(complexity)는 트라이 상에서의 노드 수, 즉 노드를 구성하는 엔트리 수로서 평가되므로 $O(2NW)$를 갖는다. 또한 검색 시간의 복잡도는 검색 과정에서의 메모리 접근 횟수와 연관되기 때문에 $O(W^2)$를 갖는다.

계층적 트라이 구조는 가장 높은 우선순위를 갖는 룰을 검색하기 위해 하위 트라이에서 상위 트라이로의 역추적이 불가피하고, 그에 따라 메모리 접근 횟수가 늘어나 검색 속도 성능을 저하시킨다.

### 2.3 셋-프루닝 트라이(Set-Pruning Trie)

셋-프루닝 트라이[4]는 계층 트라이에서 역추적에 의한 검색의 비효율성을 제거함으로써 검색 성능을 높이기 위해 제안된 구조로, 계층 트라이의 상위 트라이에서 인클로저 프리픽스 노드에 속하는 룰 정보를 인클로즈드 프리픽스 노드에 연결된 하위 트라이에 복사하여 저장하는 구조이다. 인클로저(enclosure)는 자신을 프리픽스로 하는, 즉 서브 스트링(sub-string)으로 갖는 다른 프리픽스가 같은 데이터 집합 내에 최소한 하나는 존재하는 프리픽스이고, 인클로즈드(enclosed)는 인클로저 프리픽스를 서브 스트링으로 갖는 프리픽스이다. 디스조인트(disjoint) 프리픽스는 인클로저도 인클로즈드도 아닌 프리픽스이다. 예를 들어 1101*와 110100*, 001*의 프리픽스를 고려할 때, 110100*이 1101*을 서브 스트링으로 가지므로 1101*이 인클로저, 110100*이 인클로즈드이며 001*는 1101* 및 110100*과 네스팅 관계가 없으므로 디스조인트 프리픽스이다.

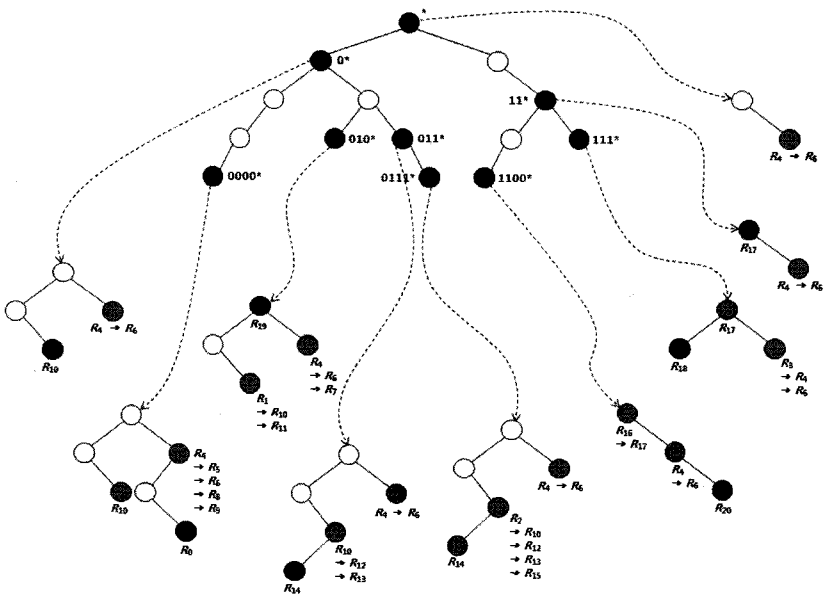



그림 2. 셋-프루닝 트라이
Fig. 2. Set-Pruning Trie

그림 2에 그림 1의 계층 트라이에 셋-프루닝을 적용하여 구성한 셋-프루닝 트라이를 보였으며, 복사된 룰이 저장된 프리픽스 노드는 회색으로 나타내었다. 셋-프루닝 트라이는 상위 트라이에서 최장 길이 검색을 통해 *BMP(best matching prefix)*를 검색하고, 이에 연결된 하위 트라이에서 우선순위가 가장 높은 룰을 찾음으로써 검색이 완료된다. 따라서 상위 트라이에서 일치된 모든 노드마다 하위 트라이로 내려가 검색이 진행될 필요가 없기 때문에 길이 검색을 통해 *BMP(best matching prefix)*를 검색하고, 이에 연결된 하위 트라이에서 우선순위가 가장 높은 룰을 찾음으로써 검색이 완료된다. 따라서 상위 트라이에서 일치된 모든 노드마다 하위 트라이로 내려가 검색이 진행될 필요가 없기 때문에 역추적이 제거된다. 위와 같은 검색 과정을 통해 검색 성능은 $O(W^2)$에서 $O(2W)$로 향상되었으나 인클로저 노드의 하위 트라이가 인클로즈드 노드의 하위 트라이에 모두 복사되어 있기 때문에, 많은 메모리가 추가적으로 사용되어 메모리 요구량이 급격히 증가한다.

## Ⅲ. 스마트 셋-프루닝 트라이 (Smart Set-Pruning Trie)

기존의 셋-프루닝 트라이의 경우, 각 상위 트라이에서 인클로져 프리픽스와 연결된 하위 트라이의 모든 노드를, 인클로즈드 프리픽스에 연결된 하위 트라이에 복사하여 저장하기 때문에, 근원지 프리픽스의 길이가 짧을수록 많은 하위 트라이에 중복되어 복사되므로 메모리 요구량이 증가한다. 예를 들어 그림 1 에서 상위 트라이의 루트 노드는 모든 프리픽스의 인클로져로서, 루트 노드에 연결된 하위 트라이에 있는 $R_4$와 $R_6$는 그림 2의 모든 하위 트라이에 복사된 것을 볼 수 있다. 또 다른 예로, 그림 1의 상위 트라이의 0* 노드에 연결된 하위트라이에는 $R_{10}$이 존재하며, 이 룰은 그림 2에서 프리픽스의 첫 비트값이 0인 모든 프리픽스에 연결된 하위 트라이로 복사된 것을 볼 수 있다.

이와 같은 특징으로부터 인클로져 프리픽스의 하위 트라이들은 인클로즈드 노드의 하위 트라이에 이미 포함되어 있으므로, 각 패스에 대하여 길이가 가장 긴 프리픽스에 연결된 하위 트라이만을 남기고 나머지 모든 하위 트라이를 제거할 수 있고, 인클로져 노드의 하위 트라이의 정보를 길이가 가장 긴 프리픽스의 하위 트라이에 포함시켜 저장할 수 있다는 추론을 할 수 있다. 제안하는 스마트 셋-프루닝 트라이는 인클로저 노드에 연결된 하위 트라이를, 상위 트라이의 각 패스상에 길이가 가장 긴 노드에 연결된 하위 트라이로 합병하는 트라이 구조를 사용하여 메모리 비효율성을 개선한 구조이다.

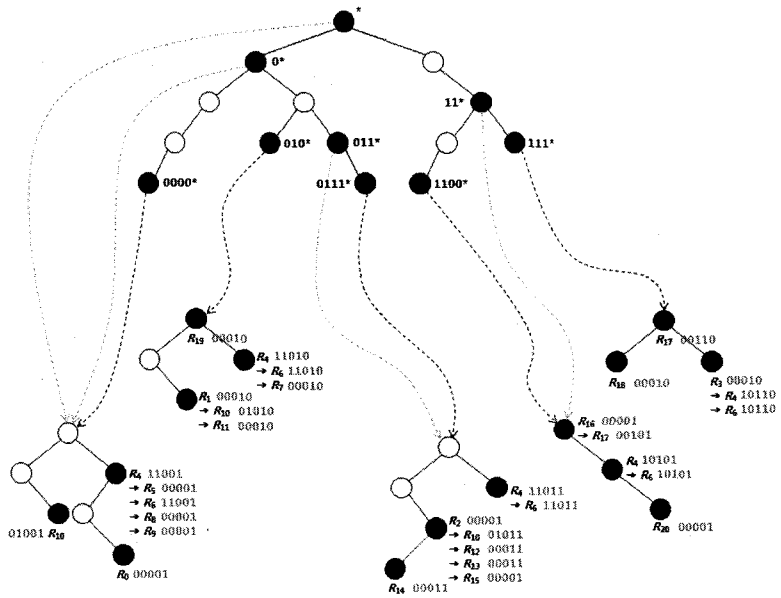



그림 3. 스마트 셋-프루닝 트라이
Fig. 3. Smart Set-Pruning Trie

그림 3에 표 1의 분류 테이블을 예로 구성된 제안하는 스마트 셋-프루닝 트라이를 나타냈으며, 변경된 포인터를 회색으로 표현하였다. 그림 3의 제안하는 스마트 셋-프루닝 구조의 경우 하위 트라이의 개수는 상위 트라이의 리프 노드의 수와 같은 것을 알 수 있다. 그림 2와 그림 3을 비교하면 하위 트라이의 개수가 9개에서 5개로 줄어든 것을 볼 수 있다.

### 3.1 트라이 합병

그림 4에 표 1의 분류 테이블에서 근원지 프리픽스가 111*인 노드의 하위 트라이와 이의 인클로저 프리픽스인 노드들의 하위 트라이 합병 과정을 표현하였다. 제안하는 구조에서는 111*의 인클로저 프리픽스인 *와 11*에 연결되어 있던 하위 트라이를 삭제하고 이 트라이들의 포인터들을 111*의 하위 트라이에 연결하였음을 알 수 있다.

인클로저 프리픽스 노드에 연결되어 있던 하위 트라이를 구성하던 룰에 관한 정보들은 모든 인클로즈드 프리픽스 노드에 연결된 하위 트라이에 프리픽스 벡터를 통하여 표현할 수 있고, 상위 트라이의 제거된 포인터를 합병된 하위 트라이 중 어떤 하위 트라이로 연결하여도 무방하나, 본 논문에서는 가장 길이가 긴 근원지 프리픽스 노드에 연결된 하위 트라이에 연결하였다.

### 3.2 프리픽스 벡터(Prefix Vector)

그림 3과 그림 4에서, 하위 트라이의 각 룰 옆에 각 룰에 해당하는 프리픽스 벡터[10]를 회색으로 나타내었다. 프리픽스 벡터는 하위 트라이의 각 노드에 저장된 룰의 $F_1$ 프리픽스 길이에 관한 정보를 벡터로 나타낸 것으로서, 프리픽스 벡터의 첫 번째 비트는 길이 0부터 시작하고, 그림 1의 예에서는 가장 긴 프리픽스의 길이가 4이므로 5비트를 갖는다. 먼저 프리픽스 벡터의 필요성을 설명한다. 제안하는 스마트 셋-프루닝 구조는 인클로져 노드의 모든 하위 트라이를 제거하기 때문에 인클로져 노드의 하위 트라이가 가지고 있던 정보를 리프 노드의 하위 트라이에 프리픽스 벡터의 형태로 저장하여야 한다.

예를 들어 그림 4에서 리프 노드인 111*의 하위 트라이를 살펴보면, 상위 트라이에서 111*까지 가는 패스 상에는 길이 0, 길이 2, 길이 3의 프리픽스가 존재하고, 길이 0인 루트 노드에 해당하는 룰인 $R_4$와 $R_6$ (그림 1참조)는 모든 하위 트라이에 복사되어야 하므로 이 룰 들의 프리픽스 벡터는 10110 (길이 0, 2, 3)이 된다. 또한 길이 2인 11*에 해당하는 룰인 $R_{17}$ (그림 1 참조)은 길이 3인 노드의 하위 트라이에 복사되어야 하므로, 이 룰의 프리픽스 벡터는 00110 (길이 2와 3)이 된다. 다른 룰 들은 길이 3인 노드의 하위 트라이에만 존재하므로 프리픽스 벡터 00010을 갖는다.

같은 룰이라도 어떤 패스상에 존재하느냐에 따라 다른 프리픽스 벡터를 가질 수 있다. 예를 들어 그림 3에서 상위 트라이의 0111* 노드로 가는 패스 상에는 길이 0, 1, 3, 4인 프리픽스가 존재하므로, 0111*의 하위 트라이에 저장된 룰 $R_4$와 $R_6$의 프리픽스 벡터는 11011임을 볼 수 있다.

### 3.3 스마트 셋-프루닝 트라이 빌드

스마트 셋-프루닝 트라이의 구조는 상위 트라이와 하위 트라이, 링크드 리스트 테이블의 세 개의

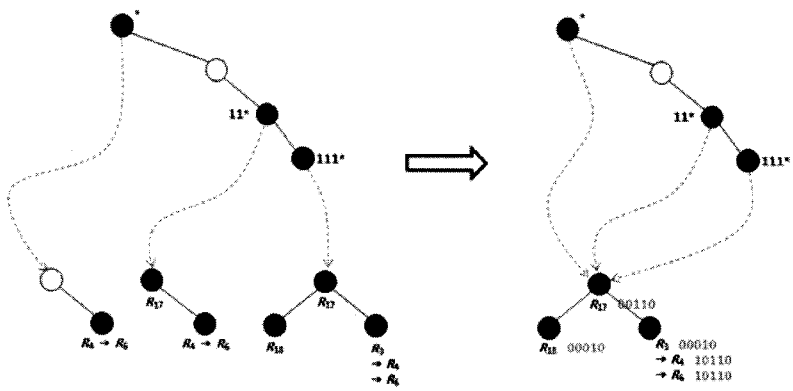



그림 4. 하위 트라이의S 합병 과정
Fig. 4. Process of Merging Lower-Level Tries

테이블에 의하여 구성되는데, 각각의 엔트리 구조를 그림 5, 6, 7에 나타내었다. 먼저 필드 1의 테이블은 근원지 프리픽스 필드에 의하여 구성되며 엔트리의 너비는 6바이트이다. 필드 1 테이블은 해당 엔트리의 유효함을 나타내는 엔트리 유효 (entry valid) 비트, 트라이의 구조를 표현하기 위한 왼쪽 및 오른쪽 자식 노드의 포인터와 하위 트라이 포인터를 갖는다. 하위 트라이 포인터의 값은 필드 2 테이블의 엔트리 주소가 된다.

필드 2 테이블은 하위 트라이 합병 후 남은 트라이, 즉 상위 트라이에서 잎 노드에 연결된 하위 트라이의 총 노드 수와 같은 수의 엔트리를 갖는다. 엔트리 유효 비트와 프리픽스 벡터 값, 근원지 포트 시작 번호, 근원지 포트 끝 번호, 목적지 포트 시작 번호, 목적지 포트 끝 번호, 프로토콜 와일드, 프로토콜 타입, 왼쪽 및 오른쪽 노드 포인터, 룰 번호, 링크드 리스트 포인터 필드로 구성되어 한 엔트리당 21바이트의 너비를 갖는다. 여기서 검색에 쓰이는 필드는 프리픽스 벡터와 근원지 포트, 목적지 포트, 프로토콜의 필드이다. 이 때, 프리픽스 벡터의 길이는 $F_1$ 필드의 모든 길이 정보에 대하여 나타낼 경우 최대 33 비트로 표현되므로 5 바이트를 차지하게 된다. 따라서 프리픽스 벡터가 차지하는 메모리 요구량을 줄이기 위해 상위 트라이에서 룰이 존재하는 중간 노드인 인클로저 노드에 대해서만 프리픽스 벡터로 나타냄으로써 메모리를 효율적으로 사용할 수 있다. 이에 대한 설명은 4절에서 자세히 다룬다. 링크드 리스트 포인터는 근원지 프리픽스와 목적지 프리픽스는 같고 포트와 프로토콜이 다른 경우에 계층적 트라이의 구조로는 같은 위치이기 때문에 식별할 수 없으므로 이를 검색할 수 있게 한 것으로써 별도로 구성된 링크드 리스트 테이블의 엔트리 주소를 나타낸다.

링크드 리스트 테이블은 엔트리 유효 비트와 프리픽스 벡터 값, 근원지 포트 시작 번호, 근원지 포트 끝 번호, 목적지 포트 시작 번호, 목적지 포트 끝 번호, 프로토콜 와일드, 프로토콜 타입, 룰 번호, 링크드 리스트 포인터 필드로 구성되어 한 엔트리당 17바이트의 너비를 갖는다. 링크드 리스트 포인터는 같은 노드에 저장된 룰들 중 가장 높은 우선순위를 가지는 하나의 룰을 제외한 나머지 룰들이 저장된 링크드 리스트 테이블의 엔트리 번호를 나타내며, 모든 일치 가능한 룰을 검색하여 우선순위가 가장 높은 룰을 검색하여야 하기 때문에 우선순위가 높은 룰부터 낮은 룰로 연결된 포인터를 이용해 검색을 할 수 있게 하였다. 하위 트라이들이 합병되는 과정에서 링크드 리스트 테이블에 삽입과 삭제가 일어나면서 테이블은 계속적으로 갱신되는데, 이 때 삭제된 룰의 경우, 엔트리 유효 비트 값을 NULL로 설정하여 엔트리를 유효하지 않게 만들 수 있다.

### 3.4 스마트 셋-프루닝 트라이에서의 검색

검색 과정은 셋-프루닝 트라이와 거의 동일하다. 필드 1 테이블에 대한 검색이 먼저 수행되며, 필드 1 테이블에 해당되는 상위 트라이에서, 입력 패킷

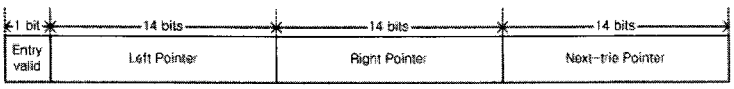



그림 5. 필드 1 테이블 엔트리 구조
Fig. 5. Entry Structure of $F_1$ Table

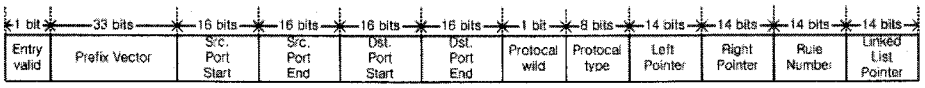



그림 6. 필드 2 테이블 엔트리 구조
Fig. 6. Entry Structure of $F_2$ Table

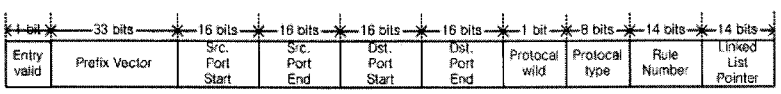



그림 7. 링크드 리스트 테이블 엔트리 구조
Fig. 7. Entry Structure of Linked List Table

근원지 주소의 MSB부터 한 비트씩 비교하며 트라이의 루트에서부터 잎 노드(leaf node)까지 0이면 왼쪽 포인터를, 1이면 오른쪽 포인터를 따라가며 이루어진다. 이 구조에서는 셋-프루닝 트라이와 마찬가지로 인클로즈드 노드의 하위 트라이에 인클로저 노드에 해당하는 룰들을 모두 복사하였기 때문에 상위 트라이에서 최장 길이 일치(longest prefix match) 검색을 수행한다.

필드 2 테이블에 대한 검색도 필드 1과 마찬가지로 입력 패킷 목적지 주소의 MSB부터 한 비트씩 비교하며 진행된다. 그림 8에 검색 과정의 의사 코드(pseudo code)를 나타내었다. 단, 하위 트라이를 빌드하는 과정에서, 메모리 효율성을 증대시키기 위하여 인클로저 노드의 길이 정보에 대해서만 프리픽스 벡터를 구성하였다. 다시 말하면, 하위 트라이는 상위 트라이의 잎 노드와 연결된 트라이이므로, 하위 트라이에 저장된 모든 룰들은 잎 노드의 검색 대상이 된다. 따라서 상위 트라이의 잎 노드에서 최장 길이 검색을 마치고 하위 트라이로 검색을 진행한 경우에는 프리픽스 벡터의 확인 과정 없이 해당 노드의 검색을 진행한다. 하지만 상위 트라이에서 잎 노드를 만나기 전에 더 이상 진행할 노드가 없어 하위 트라이 포인터를 따라 내려온 경우에는, 상위트라이에서 가장 길게 일치한 프리픽스의 하위 트라이 (그림 2 참조)에 속한 룰만을 검색하여야 한다. 즉 상위 트라이의 최장길이 일치 프리픽스의 길이에 해당하는 프리픽스 벡터의 비트 값이 셋 되어 있는 룰에 대해서만 비교한다.

먼저, query 함수에 입력 패킷 (input packet) 을 넣으면 변수 값이 초기화되면서 $F_1$ 트라이와 $F_2$ 트라이에서 순차적으로 검색을 진행하게 된다. $F_1$ 트라이 검색에서 해당 노드가 빈 노드가 아닐 경우 첫 번째 if문으로 들어가게 되며, 두 번째 if 문에서 해당 노드가 프리픽스 노드이면 $F_2$ 트라이의 포인터를 저장하고 현재 레벨(level)을 저장한다. 이 때 저장한 레벨 값은 현재까지 가장 길게 매치하는 프리픽스(longest match prefix)의 길이이며 $F_2$ 트라이 검색에서 프리픽스 벡터와의 일치 여부를 확인할 때 사용한다. 세 번째 if 문에서 해당 노드의 자식 노드 유무 여부에 따라 잎 노드인지 여부를 결정하며, 네 번째 if 문에서 만약 해당 노드가 $F_1$ 트라이의 잎 노드일 경우 $F_2$ 트라이로의 포인터를 반환하며, 다섯 번째 if 문에서 다음 비트의 값에 따라 룰 테이블의 오른쪽 또는 왼쪽 포인터를 따라간다.

$F_2$ 트라이 검색은 $F_1$ 트라이 검색에서 반환된 값

```
query(input_packet)
{
  /* variable initialize */
  F1prefix = input_packet.F1prefix;
  F1level = 0 ; //start from root level
  F1Table_address = 0 ; //root node is stored at F1table[0]
  nextTrie_ptr = -1 ; //initial value
  match_length = -1 ;
  isItLeafNode = False ; //This variable indicates that input matches
  with leaf node at F1 trie

  F1trie_lookup(F1prefix, F1level, F1Table_address, nextTrie_ptr,
  match_length, isItLeafNode);

  F2level = 0 ;
  F2root = nextTrie_ptr ;
  BMR = -1 ;
  matchWithLeafNode = isItLeafNode;
  F1nodeLength = match_length;

  F2trie_lookup(input_packet, F2level, F2root, BMR, matchWithLeafNode,
  F1nodeLength)

  return BMR;
}

F1trie_lookup(sourcePrefix, level, address, nextTrie_ptr, match_length,
isItLeafNode)
{
  isItLeafNode = 0;
  if ( table[address] is not an empty node)
  {
    if( table[address] is a prefix node)
    {
      nextTrie_ptr = table[address].nextTrie;
      match_length = level;
      if ( node has child)
        isItLeafNode = False;
      else
        isItLeafNode = True;
    }

    if( table[address] is a leaf node)
      return;
    if( next bit of prefix is '1')
      F1trie_lookup(level +1, table[address].right, nextTrie_ptr,
      match_length);
    else if( next bit of prefix is '0')
      F1trie_lookup(level +1, table[address].left, nextTrie_ptr,
      match_length);
  }
  return;
}

F2trie_lookup(input, level, address, BMR, matchWithLeafNode,
F1nodeLength)
{
  if (table[address] is not an empty node)
  {
    if(table[address] is a rule node and
      table[address].rule is higher priority than BMR and
      (matchWithLeafNode or
      table[address].prefixVector[F1nodeLength]is1))
    {
      compare the input_packet and rule;
      if (the input match with rule and rule is higher priority than
      BMR )
        BMR = table[address].rule;
    }

    if table[address] is a leaf node
      return;

    if( next bit of prefix is '1')
      F2trie_lookup(input, level +1, table[address].right,
      matchWithLeafNode, F1nodeLength);
    else if( next bit of prefix is '0')
      F2trie_lookup(level +1, table[address].left, matchWithLeafNode,
      F1nodeLength);
    }
    return;
}
```

그림 8. 검색 슈도 코드
Fig. 8. Pseudo-code for Search in Proposed Smart Set-Pruning Trie

의 포인터 주소로부터 시작하며, 해당 노드가 빈 노드가 아닐 경우 첫 번째 if 문으로 들어가게 된다. 두 번째 if 문에서 해당 노드의 룰의 우선순위가 BMR보다 높으면서, $F_1$의 잎 노드로부터 검색이 진행되었거나 $F_1$에서 저장한 길이 값에 해당하는 프리픽스 벡터의 값이 1인 조건을 모두 만족하는 경우 입력 패킷과 비교한다. 세 번째 if 문에서, 입력과 룰이 일치하는 경우 룰의 BMR을 저장한다. 만약 해당 노드가 잎 노드이면 BMR을 최종 결과 값으로 반환하며, 잎 노드가 아니면 $F_2$ 필드의 다음 비트의 값에 따라 룰 테이블의 오른쪽 또는 왼쪽 포인터를 따라간다.

예를 들어, 근원지 IP주소, 목적지 IP 주소, 근원지 포트 번호, 목적지 포트 번호, 프로토콜 타입이 $H$(0110, 0100, 67, 443, 6)인 입력 패킷에 대한 검색 과정을 그림 3을 통해 설명하면 다음과 같다. 먼저 상위 트라이에서 최장 길이 검색을 통하여 입력 패킷의 $F_1$ 필드 값인 0110과 가장 길게 일치하는 011* 노드에 도착한다. 011* 노드는 중간 노드이므로, 해당 노드의 프리픽스 길이인 3을 기억하고 하위 트라이 포인터를 따라간다. 입력 패킷의 $F_2$ 필드값이 0100 이므로, 먼저 01* 노드에 도달한다. 해당 노드에는 $F_1$ 필드 값과 $F_2$ 필드 값이 같은 여러 개의 룰이 우선순위에 따라 저장되어 있으며 상위 트라이에서 최장 일치 프리픽스 길이 3을 기억하고 있으므로, 01* 노드에 저장된 룰 중에서 길이 3에 해당하는 프리픽스 벡터 값이 1인 룰들에 대해서만 저장된 순서에 따라 비교한다. 따라서 $R_2$ 의 경우 프리픽스 벡터의 값이 길이 4인 비트에만 1값을 가지므로 비교하지 않고, 링크드 리스트 포인터를 따라 $R_{10}$으로 검색을 진행한다. $R_{10}$은 길이 3에 대한 프리픽스 벡터는 1로 설정되어 있지만, 입력 패킷과 나머지 세 필드가 일치하지 않으므로 $R_{12}$로 진행한다. $R_{12}$는 길이 3에 대한 프리픽스 벡터가 1로 설정되어 있으며, 입력 패킷과 나머지 세 필드가 일치하고, $R_{12}$ 으로부터 링크드 리스트 포인터로 연결된 나머지 룰들은 $R_{12}$ 보다 우선순위가 낮으므로, $R_{12}$을 현재까지의 BMR로 저장한 뒤 링크드 리스트 테이블을 빠져나와 하위 트라이에서의 검색을 계속 진행한다. 입력 패킷의 $F_2$ 필드의 세 번째 비트인 0을 따라 왼쪽 노드로 이동하며, 해당 노드에 저장된 $R_{14}$는 현재 BMR보다 우선순위가 낮으므로 비교하지 않고, 더 이상 진행할 수 있는 노드가 없으므로 현재까지의 BMR인 $R_{12}$을 최종 검색 결과로 출력한다.

## Ⅳ. 성능 평가

본 논문에서 제안하는 구조의 성능을 평가하기 위해 현재 인터넷 라우터에서 사용되는 룰 특성을 반영하는 ClassBench[5],[6]를 사용하여 1000개, 5000개, 10000개의 룰을 갖는 ACL(access control list), FW(firewall), IPC(IP chain)의 3개의 셋과, 해당 룰에 일치하는 인풋을 구성하여 실험하였다. 빠른 패킷분류를 위해서는 빠른 검색 속도가 반드시 요구되며 검색 속도는 입력 패킷이 속하는 클래스를 검색하는 과정에서 요구되는 메모리 접근 횟수의 영향이 가장 중요하게 평가된다[3]. 따라서 검색 성능은 BMR을 찾기 위해 필드를 검색하는데 소요되는 평균 메모리 접근 횟수($T_{avg}$)와 최대 메모리 접근 횟수($T_{max}$)로 평가될 수 있다. 또한 메모리 요구량은 테이블을 저장하는데 소요되는 총 메모리의 크기($M$)로서 평가되며 메모리 사이즈가 클수록 패킷분류를 위한 검색 시간이 많이 소모될 수 있으므로 가능한 최소의 메모리로 구성할 수 있는 알고리즘 일수록 좋은 알고리즘이라 할 수 있다.

표 2에서는 각 룰 셋에 대하여 상위 트라이에서 리프 노드를 제외한 중간 노드에 룰이 존재하는 길이 분포를 보였다. 각각의 룰 셋에 대하여 해당 길이의 프리픽스가 존재하는 경우 값을 1로 설정하여

표 2. 각 룰 셋의 중간 노드의 프리픽스 길이 정보
Table 2. Prefix Length of Internal Nodes

| Ruleset | Prefix Length of Enclosure Prefix | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
| ACL1K | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ACL5K | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ACL10K | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| IPC1K | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| IPC5K | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| IPC10K | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| FW1K | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| FW5K | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| FW10K | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

나타내었다. 이를 통해 각 룰 셋의 프리픽스가 길이별로 고르게 분포하는 것이 아니라 소수의 특정 길이에만 존재한다는 것을 알 수 있다. 특히 FW 룰 셋의 경우 리프 노드를 제외하고 최대 세 개 길이(길이 0, 1, 2)에서만 룰을 갖는다. 즉, 3.3절에서 이미 언급한 바와 같이 하위 트라이에서 33 비트의 프리픽스 벡터에 가능한 모든 프리픽스 길이 정보를 저장하는 것이 아니라, 실제로 존재하는 길이 정보만을 저장하는 프리픽스 벡터를 구성함으로써 메모리 사용량을 줄일 수 있음을 의미한다. 표 2에서 얻은 프리픽스 길이 정보를 바탕으로, 하위 트라이를 빌드할 때 프리픽스 벡터의 길이를 할당한다. 예를 들어 ACL1k의 경우는 5 비트의 프리픽스 벡터를, IPC1k는 8 비트, FW1k는 3 비트의 프리픽스 벡터를 갖게 된다.

표 3에서는 셋-프루닝 트라이와 제안하는 스마트 셋-프루닝 트라이의 상위 트라이 개수와 룰이 있는 중간 노드 즉, 인클로저 프리픽스의 길이 갯수를 나타내었으며, 인클로저 프리픽스의 길이 개수는 표 2에서 각각의 룰 셋에 대하여 값을 1로 가지는 프리픽스 원소 개수와 같다.

또한 표 4에서는 각 룰 셋에서 기존의 셋-프루닝 트라이와 제안하는 스마트 셋-프루닝 트라이의 노드 수를 비교하였다. 이 때, 기존의 셋-프루닝 트라이의 노드 수를 기준으로 하였을 때 제안하는 구조가 가지는 노드 수를 비율로 나타내었으며, 상위 트라이에서의 노드 수는 셋-프루닝 트라이와 제안하는 스마트 셋-프루닝 트라이의 경우 동일하므로 하위 트라이의 노드와 링크드 리스트 노드만으로 비교하였다. 근원지 프리픽스에 와일드카드나 짧은 프리픽스

표 3. 상위 트라이 노드 수와 중간 노드의 개수
Table 3. The Number of Nodes of Higher Tries and Distinct Lengths of Internal Nodes

| Rule | # of nodes for F1 trie | # of Enclosure Prefix length |
|---|---|---|
| ACL1k | 179 | 5 |
| IPC1k | 799 | 8 |
| FW1k | 337 | 3 |
| ACL5k | 2035 | 8 |
| IPC5k | 244 | 3 |
| FW5k | 127 | 2 |
| ACL10k | 14884 | 4 |
| IPC10k | 243 | 3 |
| FW10k | 127 | 3 |

표 4. 링크드 리스트를 포함하는 하위 트라이의 노드 수 비교
Table 4. Comparison in the Number of Nodes of Lower Tries Including Linked List

| Rule | Set-pruning | | | Smart Set-pruning | | | The Reduced Rate (%) |
|---|---|---|---|---|---|---|---|
| | Dst. | Lin. | Tot. | Dst. | Lin. | Tot. | |
| ACL1k | 12286 | 388 | 12674 | 12079 | 388 | 12467 | 98.367 |
| IPC1k | 54780 | 2693 | 57473 | 53325 | 2607 | 55932 | 97.319 |
| FW1k | 11805 | 3661 | 15466 | 11575 | 3608 | 15183 | 98.170 |
| ACL5k | 81773 | 3898 | 85671 | 79988 | 3847 | 83835 | 97.857 |
| IPC5k | 35586 | 12952 | 48538 | 34339 | 12507 | 46846 | 96.514 |
| FW5k | 31656 | 5597 | 37253 | 30588 | 5458 | 36046 | 96.760 |
| ACL10k | 352520 | 57227 | 409747 | 344575 | 55673 | 400248 | 97.682 |
| IPC10k | 35273 | 27506 | 62779 | 33854 | 26384 | 60238 | 95.952 |
| FW10k | 8592 | 8520 | 17112 | 8309 | 8326 | 16635 | 97.212 |

가 많은 경우, 이 룰들이 다른 대다수 노드의 하위 트라이들로 복사되어야 하므로, 하위 트라이의 노드 수를 증가시켜 메모리 요구량과 메모리 접근 횟수를 크게 증가시킨다. 제안하는 구조에서는 상위 트라이에서 인클로저 프리픽스 노드에 연결되어 있던 하위 트라이들을 잎 노드에 연결된 하위 트라이와 합병하였기 때문에 기존의 셋-프루닝 트라이보다 노드 수가 감소한 것을 확인할 수 있다. 룰 셋의 크기를 확장했을 때에 근원지 프리픽스로 상위 트라이의 중간 노드에 속한 룰이 많아지면서 더욱 많은 노드 수의 감소를 가져올 수 있음을 의미하며, 따라서 메모리 요구량 또한 감소할 수 있음을 의미한다.

또한 표 3에서 가장 많은 인클로저 프리픽스를 갖는 IPC1k와 ACL5k의 프리픽스 길이의 개수가 8개이므로, 프리픽스 벡터의 길이는 최대 8 비트가 되고 따라서 룰 테이블의 각 엔트리에서의 오버헤드도 최대 1 바이트로 크지 않음을 알 수 있다.

그림 9에서는 셋-프루닝 트라이의 노드 수와 제

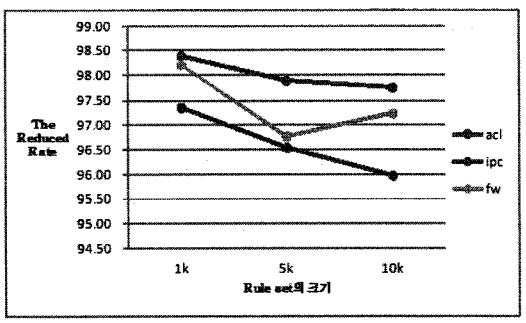



그림 9. 링크드 리스트를 포함하는 하위 트라이의 노드 수 비교
Fig. 9. Comparison in the Number of Nodes of Lower Tries Including Linked List

표 5. 제안하는 구조의 메모리 접근 횟수 비교
Table 5. The Numver of Memory Access for Proposed Algorithm

| Rule | Set-pruning | | Smart Set-pruning | |
|---|---|---|---|---|
| | Dst. | Lin. | Tot. | Dst. |
| ACL1k | 64.77 | 90 | 64.77 | 90 |
| IPC1k | 49.46 | 67 | 49.46 | 67 |
| FW1k | 36.08 | 77 | 36.28 | 77 |
| ACL5k | 64.55 | 102 | 64.56 | 102 |
| IPC5k | 56.39 | 87 | 56.47 | 87 |
| FW5k | 48.58 | 88 | 48.78 | 88 |
| ACL10k | 64.40 | 95 | 64.41 | 95 |
| IPC10k | 54.05 | 111 | 55.03 | 112 |
| FW10k | 49.45 | 96 | 49.70 | 96 |

안하는 스마트 셋-프루닝 트라이의 노드 수의 비율을 그래프로 나타내었다.

표 5 에서는 각 룰 셋에서 기존의 셋-프루닝 트라이와 제안하는 스마트 셋-프루닝 트라이의 메모리 접근 횟수를 비교하였다. 제안하는 구조에서의 메모리 접근 횟수가 기존의 셋-프루닝 트라이보다 근소한 차이를 나타나는 것을 볼 수 있는데, 이는 상위 트라이에서 인클로저 프리픽스 노드에 연결되어 있던 하위 트라이들을 잎 노드에 연결된 하위 트라이와 합병하였기 때문이다. 잎 노드의 하위 트라이에는 잎 노드의 모든 인클로저 프리픽스 노드의 하위 트라이를 포함하고 있으므로, 인클로저 프리픽스 노드의 포인터를 따라 하위 트라이에 접근하였을 경우, 불필요한 노드 접근이 발생하게 된다. 룰 셋의 크기가 증가함에 따라 중복되는 노드 수가 많아지면, 하위 트라이 합병에 따른 노드 수의 감소가 메모리 접근 횟수를 감소시켜 차이가 줄어들 것으로 예상된다.

## V. 결 론

본 논문에서는 셋-프루닝 트라이에서 하위 트라이 합병을 통하여 복사되는 노드의 수를 줄일 수 있는 스마트 셋-프루닝 패킷분류 구조를 제안하였다. 기존의 계층적 트라이는 가장 우선순위가 높은 룰을 찾기 위한 역추적이 요구되어 검색 성능이 떨어지는 한계가 있었으며, 이를 해결하기 위해 제안된 계층적 셋-프루닝 트라이는 역추적을 제거하였으나 지나치게 많은 룰 복사가 일어나 같은 룰 정보를 가지는 하위 트라이의 노드가 많다는 문제점이 있었다. 또한 그리드-오브-트라이와 같이 포인터를 사용하는 트라이의 경우 복잡한 선-계산 과정이 요구되므로 트라이의 즉각적인 업데이트가 힘들다는 단점이 있었다. 본 논문에서는 셋-프루닝 기법과 하위 트라이 합병을 통하여 룰의 복사를 제한하면서도 복잡한 선-계산이 요구되지 않는 새로운 계층적 패킷분류 알고리즘을 제안하였으며, 기존 셋-프루닝 알고리즘보다 적은 노드 수를 가지는 것을 확인하였다. 또한 룰 셋이 커질수록 복사되는 노드가 많아질 것을 예상할 수 있으므로, 그에 따라 더욱 큰 폭의 노드 수 감소가 일어날 것임을 예상할 수 있다. 노드 수의 감소는 룰 테이블에서 엔트리 수의 감소를 의미하고, 엔트리 수의 감소 폭이 커질수록 검색 성능의 향상을 기대해 볼 수 있다.

## 참 고 문 헌


[1] H. Jonathan Chao, "Next Generation Routers," Predeedings of the IEEE JPROC.2002.802001, Vol.90, Issue 9, pp.1518-1588, Sep., 2002.

[2] M. de Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkopf, "Computational Geometry: Algorithms and Applications," Springer-Verlag, 2000.

[3] Hyesook Lim and Ju Hyoung Mun, "High-Speed Packet Classification Using Binary Search on Length," IEEE/ACM ANCS 2007, Orlando, Florida, Dec. 3-4, 2007.

[4] G. Vaghese, "Network Algorithmics", Morgan Kaufmann, 2005.

[5] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel "Fast and Scalable Layer Four Switching," ACM SIGCOMM '98, Vancouver, Sep., 1998.

[6] A. Feldman, S. Muthukrishnsn, "Tradeoffs for packet classification," INFOCOM 2000, Vol.3, pp.1193-1202, Mar., 2000.

[7] F. Baboescu, S. Singh, G. Varghese, "Packet classification for core router : is there an alternative to CAMs?" IEEE INFOCOM 2003, Vol.1, pp.53-63, Mar., 2003.

[8] D. E. Taylor, J. S. Turner, "ClassBench: a packet classification benchmark," Proc.IEEE INFOCOM 2005, pp.2068 - 2079, Mar., 2005.

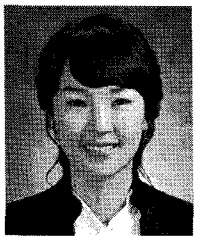
[9] D. E. Taylor, J. S. Turner, The Source Code

of Packet Classifiation Bench, http://www.arl.wustl.edu/~det3/ClassBench.index.htm

[10] H. Lim, H. Kim, and C. Yim, "IP Address Lookup for Internet Routers Using Balanced Binary Search with Prefix Vector," IEEE Trans. on Communications, Vol.57, No.3, pp.618-621, Mar., 2009

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

**민 세 원** (Seh-won Min) 준회원

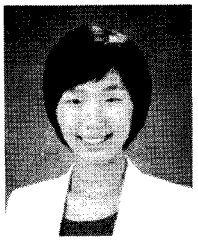


2010년 2월 이화여자대학교 전자공학과 학사

2010년~현재 이화여자대학교 전자공학과 석사과정

<관심분야> Router 및 Switch 등의 Network 관련 SoC 설계, TCP/IP 관련 하드웨어 설계

**이 나 라** (Nara Lee) 준회원



2009년 8월 이화여자대학교 컴퓨터공학·전자공학과 학사

2009년~현재 이화여자대학교 전자공학과 석사과정

<관심분야> Router 및 Switch 등의 Network 관련 SoC 설계, TCP/IP 관련 하드웨어 설계

**임 혜 숙** (Hyesook Lim) 종신회원

1986년 2월 서울대학교 제어계측공학과 학사

1986년~1989년 삼성 휴렛 팩커드 연구원

1991년 2월 서울대학교 제어계측공학과 석사

1996년 12월 The University of Texas at Austin, Electrical and Computer Engineering, Ph.D.

1996년~2000년 Lucent Technologies, Member of Technical Staff

2000년~2002년 Cisco Systems, Hardware Engineer

2002년~현재 이화여자대학교 전자공학과 교수

<관심분야> Router 및 Switch 등의 Network 관련 SoC 설계, TCP/IP 관련 하드웨어 설계