KIPS Tr. Comp. and Comm. Sys.
Vol.3, No.6 pp.179~188 pISSN: 2287-5891

http://dx.doi.org/10.3745/KTCCS.2014.3.6.179

Improving Search Performance of Tries Data Structures for Network Filtering by Using Cache

Hoyeon Kim† · Kyusik Chung††

ABSTRACT

Due to the tremendous amount and its rapid increase of network traffic, the performance of network equipments are becoming an important issue. Network filtering is one of primary functions affecting the performance of the network equipment such as a firewall or a load balancer to process the packet. In this paper, we propose a cache based tri method to improve the performance of the existing tri method of searching for network filtering. When several packets are exchanged at a time between a server and a client, the tri method repeats the same search procedure for network filtering. However, the proposed method can avoid unnecessary repetition of search procedure by exploiting cache so that the performance of network filtering can be improved. We performed network filtering experiments for the existing method and the proposed method. Experimental results showed that the proposed method could process more packets up to 790,000 per second than the existing method. When the size of cache list is 11, the proposed method showed the most outstanding performance improvement (18.08%) with respect to memory usage increase (7.75%).

Keywords : Network Filtering, Tries Structure, Search Performance Improvement

네트워크 필터링에서 캐시를 적용한 트라이 구조의 탐색 성능 개선

김 호 연† · 정 규 식††

요 약

트래픽의 엄청난 양과 함께 급격한 증가로 인하여 네트워크 장비들의 성능이 중요한 이슈가 되고 있다. 방화벽 또는 부하분산기와 같이 패킷을 처리하는 네트워크 장비에서 성능에 영향을 주는 주요한 기능 중에 하나가 네트워크 필터링이다. 본 논문에서는 네트워크 필터링의 탐색 방법 중의 하나인 기존 트라이 방법의 성능을 개선하기 위하여 캐시를 적용한 트라이를 제안한다. 클라이언트와 서버 사이의 패킷 교환에서 한 번에 다수의 패킷이 송수신되는 경우에 대하여, 기존 방법은 동일한 탐색을 반복적으로 수행한다. 반면, 본 논문에서 제안하는 방법은 기존 방법에 캐시를 적용하여 불필요한 반복 탐색을 방지함으로써 네트워크 필터링 성능이 향상될 수 있다. 기존 방법과 제안 방법을 이용한 네트워크 필터링 실험을 수행하였다. 실험결과는 제안 방법이 기존 방법에 비하여 최대 초당 790,000개의 패킷을 더 처리할 수 있었음을 보여준다. 캐시 리스트 크기가 11일 때, 메모리 사용 증가량(7.75%) 대비 성능 개선(18.08%)이 가장 우수하였다.

키워드 : 네트워크 필터링, 트라이 구조, 탐색 성능 향상

1. 서 론

최근 인터넷에서는 엄청난 양의 트래픽이 매일 발생하며, 또한 트래픽 발생량은 시간이 지날수록 계속해서 증가하고 있는 추세이다. Cisco VNI(Visual Networking Index) 2013[1]에 따르면, 2017년에 월간 발생하는 트래픽양이 121 엑사바이트(exa byte)에 달할 것으로 예상된다. 1 엑사바이트는 10^18 바이트로 미 의회도서관 인쇄물을 합친 양의 10만배에 해당하는 정보량인데 과거에는 상상도 할 수 없을 정도로 엄청난 양이다.

이러한 지속적인 인터넷 트래픽 급증 추세에 따라, 관련된 네트워크 장비들의 성능이 중요한 이슈가 되고 있다. 네트워크 장비의 성능 향상을 위하여 하드웨어적인 방법뿐만 아니라 소프트웨어적인 방법에 관한 연구가 필요하다. 소프트웨어적인 방법은 비용이 저렴하여 다방면에서 장점을 갖는다. 그렇기 때문에 네트워크 필터링 알고리즘, 패킷 I/O 처리 프레임워크 및 아키텍처와 같은 소프트웨어적인 방법을 통한 네트워크 성능개선에 관한 연구는 더욱더 많은 관

※ 이 논문은 펌킨네트웍스(주) 지원을 받아 연구되었음.
† 준 회 원 : 숭실대학교 정보통신공학과 석사
†† 정 회 원 : 숭실대학교 정보통신전자공학부 교수
Manuscript Received : January 2, 2014
First Revision : March 25, 2014; Second Revision : June 2, 2014
Accepted : June 4, 2014
* Corresponding Author : Kyusik Chung(kchung@q.ssu.ac.kr)

심을 받고 있다.

패킷을 처리하는 네트워크 장비들은 기본적으로 네트워크 필터를 갖는다. 네트워크 필터란 해당 장비가 패킷을 허용할 것인가 거부할 것인가를 결정하는 조건들의 모음을 의미한다. 네트워크 장비로 입력되는 모든 패킷에 대하여 필터링이 수행되기 때문에 필터링에 관한 성능 개선은 곧 네트워크 장비의 처리량 증가를 의미한다. 네트워크 필터링 기능에 특화되어 있는 장비로는 방화벽과 부하분산기 등이 있다.

본 논문에서는 네트워크 필터링 방법 중의 하나인 트라이(Tries)구조의 성능을 개선하기 위하여 캐시를 적용한 트라이를 제안한다. 클라이언트와 서버 사이의 패킷 교환에서 한 번에 다수의 패킷이 송수신되는 경우에 대하여, 기존 방법은 동일한 탐색을 반복적으로 수행한다. 반면, 본 논문에서 제안하는 방법은 기존 방법에 캐시를 적용하여 불필요한 반복 탐색을 방지함으로써 네트워크 필터링 성능이 향상될 수 있다. 제안방법의 성능 개선 효과를 검증하기 위해 2만 개의 필터를 적용하는 네트워크 필터링 실험을 수행하였다. 서버 30대와 클라이언트 50대로 구성된 클러스터 환경에서 SPECweb을 수행하여 실험에 사용할 트래픽 패턴을 수집하였다. 테스트 툴로는 Netmap을 사용하였다. Netmap에서 제공하는 packet generator에서 Sender 측을 수집한 트래픽 패턴을 반영할 수 있도록 수정하였고, Receiver 측을 본 논문에서 제안하는 필터링 알고리즘을 적용할 수 있도록 수정하였다.

본 논문의 구성은 다음과 같다. 2장에서는 기존에 존재하는 네트워크 필터링 알고리즘에 대하여 소개한다. 3장에서는 기존 방법인 트라이 구조의 성능을 개선한 캐시를 적용한 트라이를 설명하고, 4장에서는 실험 및 토론을, 5장에서는 결론을 제시한다.

2. 연구 배경

2.1 네트워크 필터(Network filter)

네트워크 필터란 패킷이 수신되어 어플리케이션 영역으로 전달되기 이전에 거쳐야 하는 일종의 관문이다. 네트워크 필터는 Protocol, Source IP, Destination IP, Source Port 그리고 Destination Port의 총 다섯 가지 필드로 구성된다. 모든 패킷에는 위에 언급한 다섯 필드에 해당하는 값들이 존재하기 때문에, 사용자는 필터 규칙을 생성하여 다섯 가지 필드에 대한 값을 설정함으로써 입력된 패킷을 허용하거나 거부할 수 있다[2].

2.2 기존 연구

기존의 네트워크 필터링 알고리즘은 크게 “선형 탐색”과 “트라이”가 존재 한다. 선형탐색은 설정된 필터 정보를 순차적으로 배치 및 탐색하여 필터링을 수행하는 방법이며, 트라이는 설정된 필터 정보를 이용하여 트라이 구조를 생성하고 생성된 트라이 구조를 탐색하여 필터링을 수행하는 방법이다. 패킷 분류(Packet Classification) 알고리즘을 정리한 참고문헌[2]는 Hierarchical tries, Set-pruning tries, Grid-of-tries를 소개한다. Hierarchical tries와 Set-pruning tries는 기본 데이터 구조(Basic data structures) 방식이며, Grid-of-tries는 기하학적인 알고리즘(Geometric algorithms) 방식이다.

Hierarchical tries는 d-차원 계층 기수(radix) 트라이를 의미하며, 가장 기본적인 트라이 방식이다. 다차원 트라이의 트라이 구조는 1차원 트라이의 간단한 확장으로서 prefix 규칙에 따라 재귀적으로 구성되며, 각 계층에는 다음계층으로 연결되는 포인터가 존재한다. 입력된 패킷 정보에 따라 prefix 규칙이 적용되어 1차원부터 d차원까지 탐색하여 필터링을 수행한다. Hierarchical tries는 “multi-level tries”, “backtracking-search tries”, “trie-of-tries” 라고도 불린다.

Set-pruning tries[3][4]는 Hierarchical tries와 유사하지만 트라이구조를 생성할 때 필터 규칙을 복제하여 중복된 내용을 포함시킨다. 저장 공간을 더 많이 소비하여 쿼리시간(query time)을 줄일 수 있도록 고안된 방식이다.

Grid-of-tries[5][6]는 필터 규칙 할당에 따른 저장 공간을 줄이기 위해 고안된 2차원 분류 방식이다. 또한 쿼리시간을 단축시키기 위하여 스위치 포인터를 도입하였다.

선형 탐색을 적용한 대표적인 네트워크 필터는 리눅스에서 기본적으로 제공하는 “iptables”가 있으며, 트라이의 대표적인 예는 리눅스를 기반으로 동작하는 “nf_hipac”이 있다.

1) 선형 탐색(Linear search)

선형 탐색[7][8]은 사용자에 의하여 설정된 필터 규칙을 순차적으로 저장 및 관리한다. 단순하게 필터 규칙을 저장하므로, 필터의 저장 및 관리와 관련된 오버헤드가 적다. 하지만, 순차적으로 설정된 필터 규칙을 하나씩 비교해 나가기 때문에, 필터 규칙의 수와 패킷 처리량은 반비례 한다[9].

선형 탐색은 패킷이 입력되었을 때, 입력된 패킷의 모든 필드(Protocol, Source IP, Destination IP, Source port, Destination. port) 값이 설정된 필터의 값에 해당되는지 여부를 필터가 저장된 순서대로 각각 비교한다. 패킷의 모든 필드가 설정된 필터의 내용에 해당할 경우에 매치되며, 마지막으로 저장된 필터의 내용까지 확인하여 해당되지 않을 경우 비매치 된다. 즉, 선형 탐색은 설정된 필터 규칙에 해당되지 않는 패킷이 입력되었을 경우, 저장된 모든 필터들의 규칙을 확인해야만 입력된 패킷의 비매치를 판단할 수 있다.

Table 1. User filter setting example

Filter	Src. IP Range	Dst. IP Range	Dst. Port Range
Filter_0	[3, 8]	[0, 15]	[3, 11]
Filter_1	[0, 8]	[0, 8]	[3, 15]
Filter_2	[3, 15]	[0, 15]	[0, 11]
Filter_∞	[0, 15]	[0, 15]	[0, 15]

선형 탐색의 동작을 예를 들어 설명한다. 예는 Source IP, Source port, Destination port 세 가지 필드만을 고려한다. Table 1의 필터 설정을 가정하고, 입력되는 패킷은 Src. IP=4, Src. Port=6, Dest. Port=2를 가정한다.

패킷 매치 과정을 설명하면 다음과 같다.

⓪ 패킷 수신.
① Filter_0에 접근. 각각 필드 비교.
② 패킷의 Dst. Port=2이므로, Filter_0의 조건에 해당하지 않음.
③ Filter_1에 접근. 각각 필드 비교.
④ 패킷의 Dst. Port=2이므로, Filter_1의 조건에 해당하지 않음.
⑤ Filter_2에 접근. 각각 필드 비교.
⑥ 패킷의 모든 필드가 Filter_2의 조건에 해당됨. 매치.

Fig. 1은 패킷을 수신하여 사용자의 필터 설정과 비교하는 과정을 나타낸 그림이다.

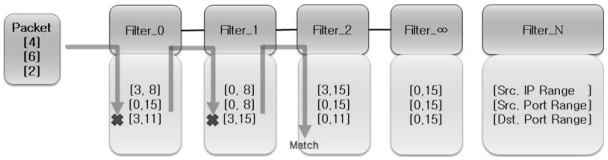

Fig. 1. Linear search process example

2) 트라이(Tries)

트라이[10]는 사용자에 의하여 설정된 필터 규칙을 이용하여 트라이 구조를 생성한다. 트라이 구조는 설정된 필터의 각 필드를 디멘션(Dimension)으로 나누어 저장한다. 설정된 필터들의 Src. IP 내용으로 디멘션_1이 생성되며, Src. Port 내용으로 디멘션_2가 생성되는 방식이다. 필터 설정에 따라서 각 디멘션에는 하나 혹은 다수의 노드(Node)가 존재한다. 각 노드는 필터의 설정 값들의 최소값과 최대값으로 전체 범위가 결정되며, 필터의 설정에 따라 영역이 분리된다. 분리된 각각의 영역을 노드엔트리(Node Entry)라 한다. 각각의 노드엔트리는 인덱스(Index)와 값(Value)으로 구성된다. 값은 설정된 필터의 내용으로 크기가 작은 순서부터 정렬하여 노드엔트리들에 저장되며, 인덱스는 각 값에 부여된 순서이다. 마지막 디멘션의 노드들을 제외한 나머지 디멘션의 노드들은 각 노드엔트리에 다음 디멘션에 해당하는 하위 노드에 연결되며, 마지막 디멘션의 노드들은 각 노드엔트리에 필터가 연결된다. Fig. 2는 Table 1을 기반으로 생성된 트라이 구조를 나타낸 그림이다.

a) 트라이의 노드 구조: Fig, 2.에서 사각점선 안에 Node 1-1을 예로 트라이의 노드에 대해 설명한다. 그림에서 Index1(0~2), Index2(3~7), Index3(8~15)의 세 구간으로 나뉘어 있는 것을 확인할 수 있다. 이는 표 1에서 네 개의 필터의 Src. IP Range값을 구간별로 구분한 결과이다. 각 구간은 필터의 설정 규칙에 따라 나뉜다. “Index1” 구간은 필터1과 필터∞의 설정이 해당되고 Node 2-1에 연결되며, “Index2” 구간은 필터0, 필터1, 필터2, 필터∞의 설정이 해당되고 Node 2-2에 연결되며, “Index3” 구간은 필터2, 필터∞의 설정이 해당되고 Node 2-3에 연결된다. 마지막 Dimension의 노드들은 Index구간에 매치되는 필터가 연결된다. 해당되는 필터가 여러 개일 경우, 우선순위가 가장 높은 필터가 선택된다.

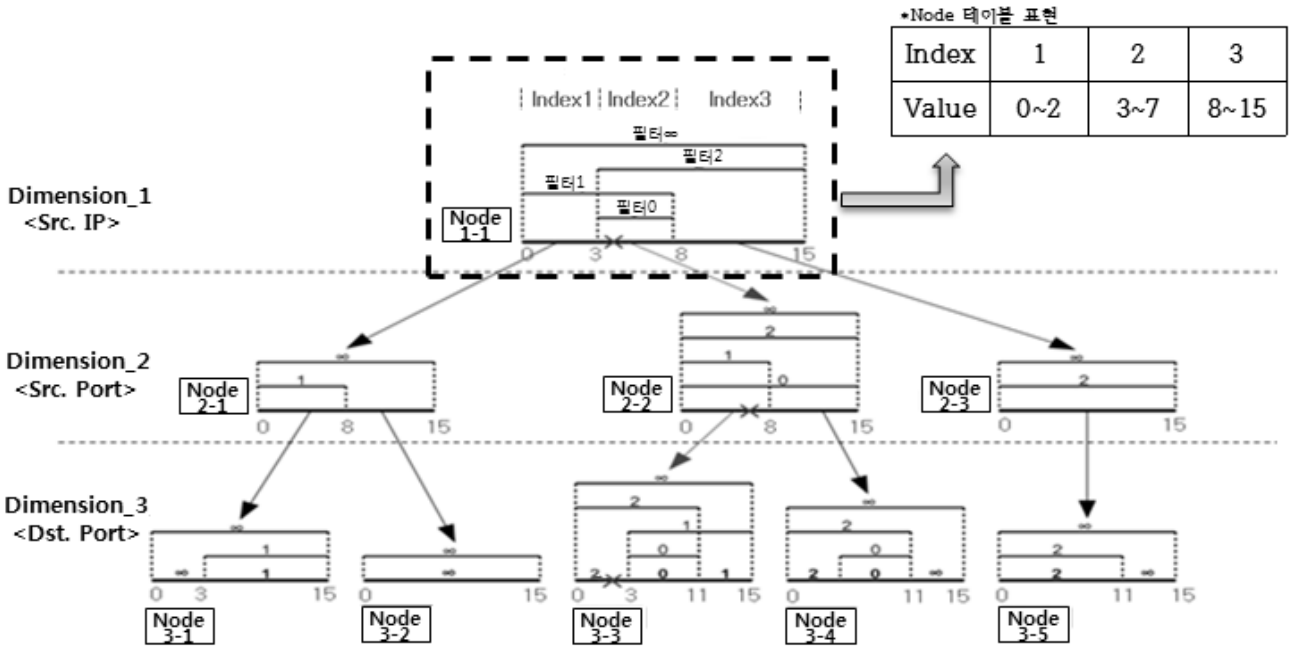

Fig. 2. Tries structure

b) 트라이의 이진 탐색: Fig. 3은 기존 트라이의 노드 탐색 방법을 나타낸 그림이다. “Node Table”의 Value_* 들은 크기순으로 정렬되어 있다. Packet의 “Value=Value_2”를 가정하여 탐색 과정을 설명하면 다음과 같다.

⓪ 패킷 수신.
① Node에 접근. 이진탐색 수행하여 Index_4에 접근.
② Index_4의 Value_4와 패킷의 Value_2 비교. Value_2는 Value_4보다 작음.
③ 계속해서 이진 탐색 수행. Index_2에 접근.
④ Index_2의 Value_2와 패킷의 값 비교. 값이 같으므로 매치.

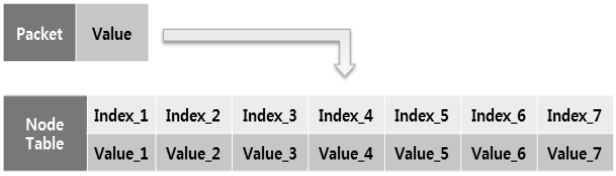

Fig. 3. Tries's search process

c) 트라이의 탐색 예: 입력되는 패킷이 Src. IP=4, Src. Port=6, Dest. Port=2일 때, Table 1의 내용을 기반으로 생성된 Fig. 2를 참고하여 설명하면 다음과 같다.

⓪ 패킷 수신.
① Node1-1에 접근. 이진 탐색을 통하여 Index=2 탐색.
② 패킷의 Src. IP=4이므로, Index=2에 해당.
③ Node1-1의 Index=2에 연결된 Node2-2에 접근. 이진 탐색을 수행하여 Index=1 탐색.
④ 패킷의 Src. Port=6이므로, Index=1에 해당.
⑤ Node2-2의 Index=1에 연결된 Node3-3에 접근. 이진 탐색을 수행하여 Index=2 탐색.
⑥ 패킷의 Dst. Port가 Index=2에 해당하지 않으므로. 이진 탐색을 수행하여 Index=1 탐색.
⑦ 패킷의 Dst. Port=2이므로, Index=1에 해당.
⑧ Node3-3은 마지막 디멘션의 노드이므로, 입력된 패킷은 Node3-3의 Index=1에 연결된 Filter_2에 매치.

3) 선형탐색 vs. 트라이

선형 탐색을 적용한 “iptables”와 트라이 구조를 적용한 “nf_hipac”의 성능을 비교한 기존의 연구가 있다[11]. 아래 Fig. 4는 “nf_hipac”[1]과 “iptables”[8]의 성능을 비교한 그래프이다. 필터의 수에 비례하여 트라이 구조의 성능이 더 우수함을 확인할 수 있다.

2.3 본 논문의 접근 방식

본 논문에서는 기존 트라이의 성능을 개선하기 위하여 캐시를 적용한 트라이 구조를 제안한다. 캐시는 최근에 매치된 필터들에 관련된 정보를 저장하며, 노드 상에서 이진 탐색을 수행하기 이전에 먼저 탐색된다.

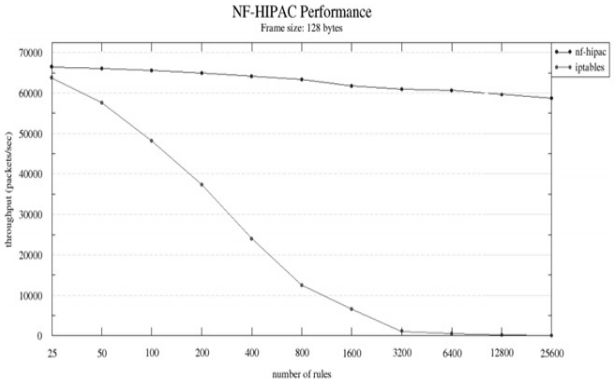

Fig. 4. Comparison of performance between “nf_hipac” and “iptables”

클라이언트와 서버 사이의 패킷 교환에서 한 번에 다수의 패킷이 연속적으로 송수신되는 경우에 대하여, 일정량의 캐시 히트를 기대할 수 있다. 제안하는 방법은 캐시 히트율에 비례하여 기존 방법보다 성능이 개선될 수 있다.

3. 캐시를 적용한 트라이

3.1 제안 방법의 구조

본 논문에서는 기존 트라이 구조의 탐색 성능을 개선하기 위하여 트라이 구조 내의 각 노드에 캐시리스트를 추가하는 방식을 제안한다. Fig. 5는 Table 1을 기반으로 생성된 본 논문에서 제안하는 캐시를 적용한 트라이 구조를 나타낸다. 2장에서 나타낸 기존 트라이구조와 비교 해보면 Fig. 2에서 각 노드에 캐시리스트(Cache List)가 추가된 것을 확인할 수 있다.

제안하는 방법의 캐시리스트에 관한 특성은 다음과 같다.

- 캐시리스트는 각각의 노드(Node)마다 존재한다.
- 캐시리스트는 최근에 매치된 노드엔트리의 인덱스(Inde-x)를 저장한다.
- 캐시리스트는 오름차순으로 정렬하여 값을 저장한다.

3.2 제안 방법의 동작 과정

Fig. 6은 Fig. 5를 기반으로, 제안하는 방법이 필터를 매치하는 과정을 전체적으로 나타내는 그림이다.

각 디멘션의 노드를 탐색할 때, 우선적으로 캐시 리스트를 통하여 노드에 접근하는 것을 확인할 수 있다. 그림에는 ①, ②, ③, ④의 탐색 예가 있다. ①은 최종적으로 필터 매치가 발생한 경우이다. ②는 Dimension_1에서 비매치가 발생한 경우이며, ③은 Dimension_2에서 비매치가 발생한 경우이고, ④는 Dimension_3에서 비매치가 발생한 경우이다.

Fig. 7은 제안 방법이 노드를 탐색하는 과정을 간략하게 나타낸 그림이다. 패킷이 입력되어 캐시리스트(Cache list)를 통하여 탐색하는 Step[1]과 노드테이블(Node table)에서 직접 탐색하는 Step[2]의 두 가지 과정으로 탐색을 수행하는 것을 확인할 수 있다.

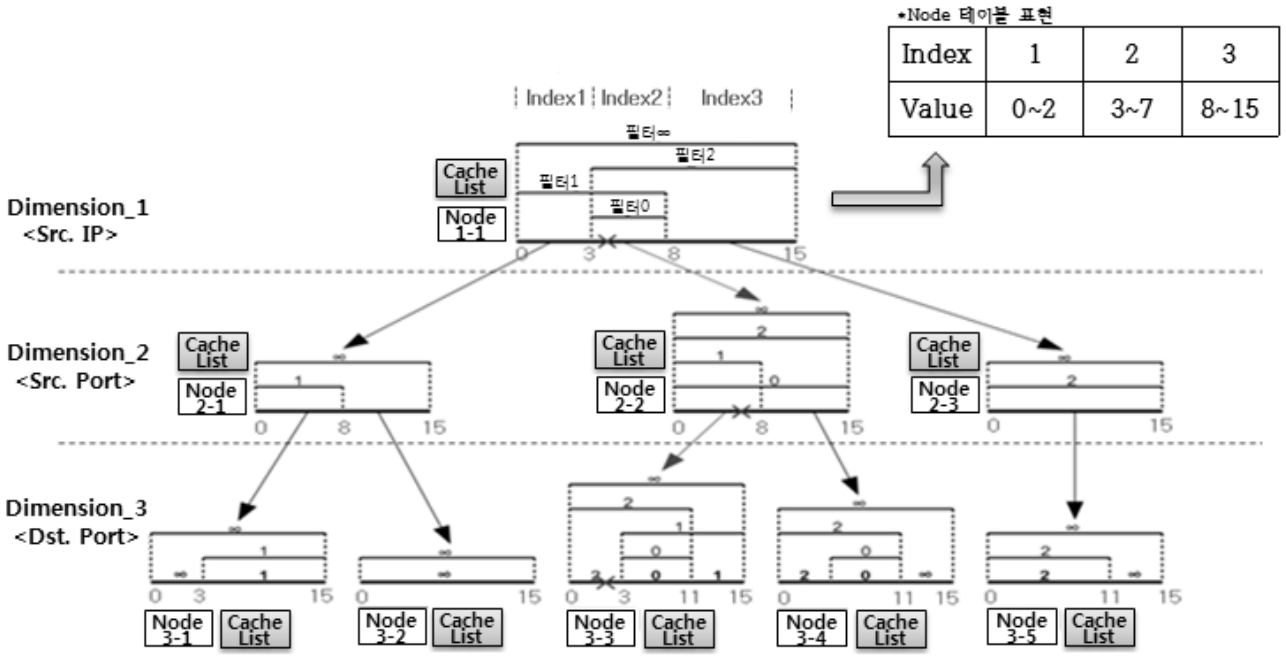

Fig. 5. Cache based Tries structure

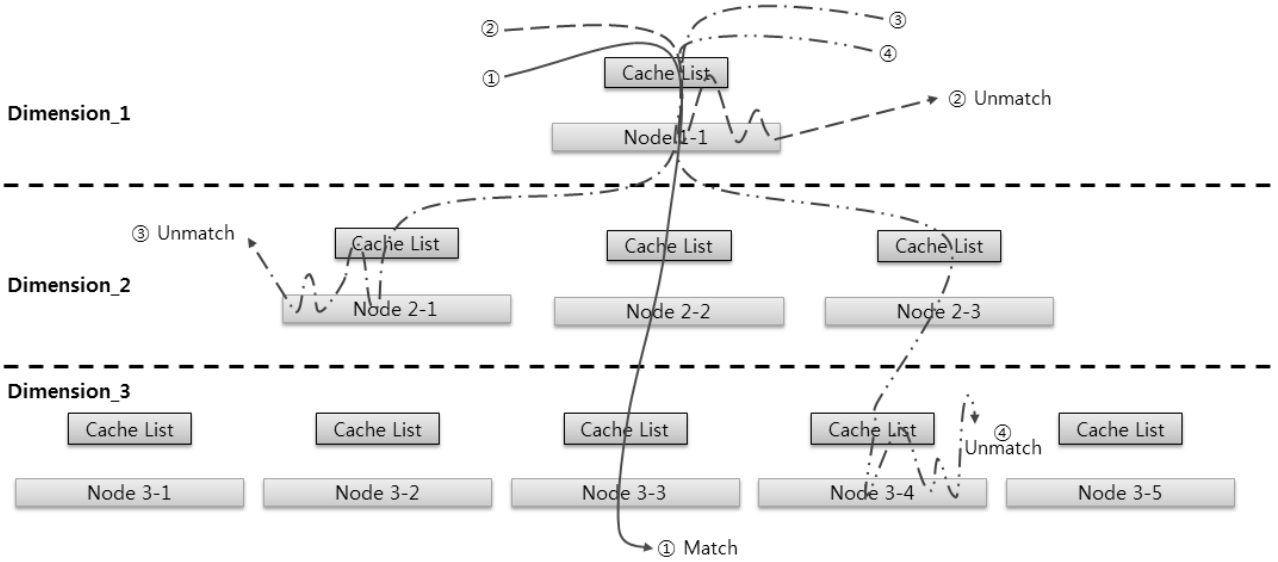

Fig. 6. Total search process of cache based Tries

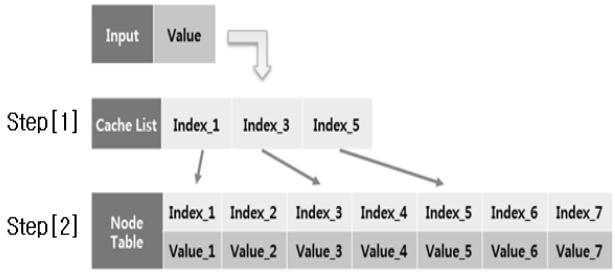

Fig. 7. Search process of cache based Tries

1) 탐색

제안하는 방법은 패킷이 입력되었을 때, 첫 번째 디멘션(Dimension)의 노드(Node)에 접근한다. 우선적으로 해당 노드의 캐시리스트에 접근하여 캐시리스트에 저장된 인덱스를 확인한다(Step[1]). 그 후, 캐시리스트에서 확인한 인덱스에 해당하는 노드엔트리의 값과 패킷의 값을 비교한다. 만약, 패킷의 값이 노드엔트리의 값에 해당 된다면(캐시히트에 해당), 노드엔트리에 연결된 다음 디멘션의 노드에 접근하여 동일하게 탐색을 계속하며, 패킷의 값이 노드엔트리의 값에 해당되지 않는다면 다른 캐시리스트에 접근하여 탐색을 계

속한다. 모든 캐시리스트에서 패킷의 값이 해당하는 노드엔트리를 찾지 못할 경우(캐시미스에 해당), 이진 탐색을 수행하여 노드의 탐색을 계속한다(Step[2]). 이때, 이진탐색의 범위는 앞서 탐색한 캐시리스트의 값들을 기준으로 설정된다. 이진 탐색을 계속 수행하면서 패킷의 값이 노드엔트리의 값에 해당하는 경우가 존재한다면 노드엔트리에 연결된 다음 디멘션으로 이동하여 위에서 설명한 탐색(Step[1]과[2])을 계속한다. 패킷의 모든 필드 값이 각 디멘션의 노드엔트리에 해당되었다면 필터 매치가 발생한다. 마지막 디멘션의 노드엔트리들에는 다음 디멘션의 노드가 아닌 매치되는 필터가 연결되어 있다.

만약, 이진 탐색을 계속 수행하였는데도 불구하고 패킷의 값이 해당되는 노드엔트리를 찾을 수 없다면, 각 디멘션에서 필터 비매치가 발생하여 입력된 패킷은 폐기된다.

Fig. 8은 앞서 설명한 제안 방법의 탐색 과정을 흐름도로 나타낸 것이다.

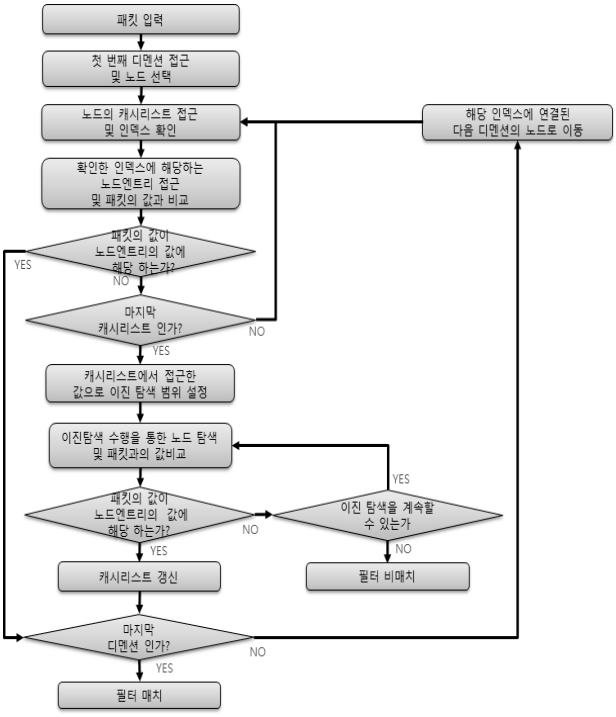

Fig. 8. Flow chart of cache based Tries

2) 캐시리스트 갱신

제안하는 방법은 캐시리스트(Cache List)에 인덱스(Index)를 저장할 때, 오름차순으로 정렬하여 저장한다. 또한, 캐시리스트를 갱신할 때, 새롭게 저장될 인덱스는 기존에 캐시리스트에 정렬된 상태로 저장되어 있는 인덱스들과 크기를 비교하여 위치를 결정하고, 결정된 위치에서 가장 인접한 두 개의 인덱스와 저장될 새로운 인덱스의 차이를 비교하여 차이가 작은 후보를 교체한다. 이러한 방법은 캐시 미스가 발생할 경우 탐색해야 하는 이진 탐색의 범위를 균등하게 하는 효과가 있다. 예를 들면, Fig. 9에서 캐시리스트에 30, 50, 80이 저장되어 있다고 하자. 새로운 52가 추가되었을 때 이를 캐시리스트에 갱신하는 방식이 두 가지가 있는데, 본 논문에서는 ⓐ 방식으로 캐시리스트를 갱신하여 캐시리스트 내 각 값들의 간격을 균등하게 유지하도록 한다.

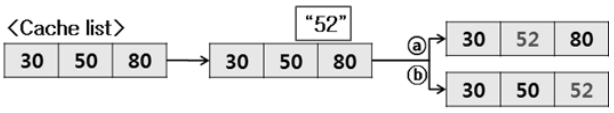

Fig. 9. "Cache List" renewal

4. 실험 및 토론

4.1 실험에 적용할 트래픽 패턴

본 논문에서는 SPECweb을 통하여 클라이언트와 서버 사이의 트래픽 교환 패턴을 수집하여 실험에 적용한다.

1) SPECweb

SPECweb[12]은 웹서버 성능 측정에 널리 사용되는 툴이다. SPECweb은 Banking, E-commerce, Support 시나리오를 제공한다. 각각 시나리오는 사용자가 인터넷뱅킹, 웹쇼핑, 파일다운로드와 관련한 동작을 시뮬레이션 한다. 예를 들어, SPECweb의 Banking 시나리오는 사용자가 로그인하여 계좌를 확인하고 이체를 하는 등의 동작을 수행한다. 실제로 SPECweb에서 사용되는 서버들에는 각 시나리오에 대한 다양한 종류 및 크기의 페이지들이 존재하며, 일반적인 서버 측정 툴과 다르게 Besim이 추가되어 사용자의 이용패턴을 시뮬레이션할 수 있다. 즉, SPECweb에서 발생되는 트래픽 교환은 실제 클라이언트와 서버 사이의 트래픽 교환과 흡사하다[13].

본 논문에서는 일반적인 트래픽 교환 패턴에서 제안하는 방식의 성능 개선 정도를 파악하기 위하여, SPECweb의 Banking시나리오를 적용하여 서버와 클라이언트 사이의 트래픽 패턴을 수집한다. 수집된 트랙픽 패턴은 네트워크 레벨 실험에 적용한다.

2) SPECweb 트래픽 수집 환경

Table 2는 SPECweb 구성에 사용된 하드웨어를 정리한 표이다.

Fig. 10은 SPECweb 트래픽을 수집하기 위한 SPECweb 구성도이다.

3) SPECweb 트래픽 수집

Load Balancer에서 "TCPdump"[14]를 이용하여 클라이언트들과 서버들 사이에 교환되는 모든 패킷을 수집한다. 이중에서 네트워크 필터링이 적용되는 클라이언트의 요청 부분만을 고려하고 서버로부터의 응답은 생략한다. "TCPdump"로부터 수집한 패킷정보 중에서 네트워크 필터링에 사용되는

Table 2. Hardware used at SPECweb Configuration

	CPU	RAM (MB)	Software	Note
Client1	Intel(R) Core(TM) i5-3550 CPU @ 3.3GHz	8192	SPECweb - Client	
Client2	Intel(R) Core(TM) i5-3550 CPU @ 3.3GHz	8192	SPECweb - Client	
Client3	Intel(R) Core(TM) i5-3550 CPU @ 3.3GHz	8192	SPECweb - Client	
Client4	Intel(R) Core(TM) i5-3550 CPU @ 3.3GHz	8192	SPECweb - Client	
Client5	Intel(R) Core(TM) i5-3550 CPU @ 3.3GHz	8192	SPECweb - Client	
VM-1	Intel(R) Core(TM)2 Quad CPU @ 2.5GHz	2048	SPECweb - Client	VM Client6~10
VM-2	Intel(R) Core(TM)2 Quad CPU @ 2.5GHz	2048	SPECweb - Client	VM Client11~15
VM-3	Intel(R) Core(TM)2 Quad CPU @ 2.5GHz	2048	SPECweb - Client	VM Client16~20
VM-4	Intel(R) Core(TM)2 Quad CPU @ 2.5GHz	2048	SPECweb - Client	VM Client21~25
VM-5	Intel(R) Core(TM) i7-2600k CPU @ 3.4GHz	8192	SPECweb - Client	VM Client26~38
VM-6	Intel(R) Core(TM) i5-3570 CPU @ 3.4GHz	8192	SPECweb - Client	VM Client29~50
Besim	Intel(R) Core(TM) i5-3550 CPU @ 3.3GHz	8192	SPECweb - Besim	
Server 1~30	Intel(R) Core(TM) i5-3550 CPU @ 3.3GHz	8192	SPECweb - Server	Cluster of 30 servers

Protocol, Source IP, Source port, Destination IP, Destination port 필드 정보만을 사용한다.

4.2 실험 환경

위에서 수집한 트래픽 패턴을 네트워크 환경에 적용하여, Netmap 패킷 I/O 프레임워크[15] 환경에서 실험을 수행한다. Netmap은 패킷 I/O 프레임워크로서 커널을 경유하지 않고, 디바이스 드라이버에서 어플리케이션으로 직접 패킷을 전달하는 방식으로 동작한다[16][17]. 실험에서는 초당 패킷 처리량(packet per sec.)을 측정하며, 측정 툴로서 Netmap에서 제공하는 pkt-gen(packet generator)을 사용한다. pkt-gen

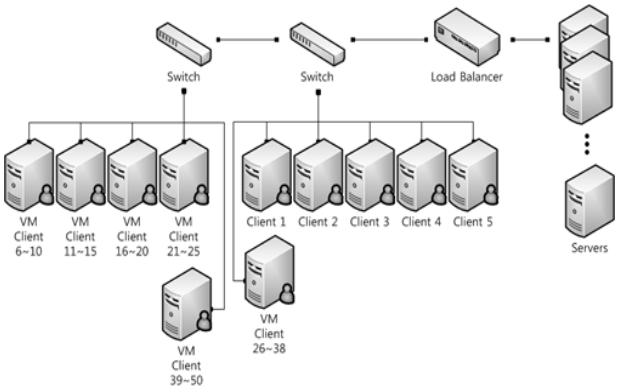

Fig. 10. SPECweb Configuration

은 C언어로 작성된 어플리케이션이며 Sender 모드와 Receiver 모드로 동작한다. Sender 측은 수집한 트래픽 정보를 반영할 수 있도록 수정하였으며, Receiver 측은 필터링 알고리즘을 적용할 수 있도록 수정하였다.

1) 실험 방법

Sender는 계속하여 패킷을 송신하고, Receiver는 수신한 패킷의 필터링을 수행한다. 실험은 PPS(Packet Per Sec.) 측정을 목적으로 하기 때문에, 필터링을 완료한 패킷은 폐기 한다. 4장에서 수집한 트래픽을 적용하였으며, 필터 설정은 2만 개를 사용하였다.

Table 3은 실험에 사용한 하드웨어 사양을 정리한 표이고, Fig. 11은 실험의 구성도이다.

Table 3. Hardware used at experiment

	CPU (Hz)	RAM (MB)	NIC	OS	Test Tool	Software
Sender	Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz	8192	Intel x520 10gi gabit NIC	Ubuntu 12.10	Netmap pkt-gen (Sender)	
Receiver	Pentium(R) Dual-Core CPU E6700 @ 3.20GHz	2048			Netmap pkt-gen (Receiver)	Applying filtering algorithm

Fig. 11. Experiment Configurations

4.3 실험 결과 및 분석

Fig. 12는 캐시 리스트 크기별로 성능을 측정한 필터링 처리량(Mpps)에 대한 실험 결과이다.

캐시를 사용하지 않는 기존 트라이 방법의 처리량은 3.90Mpps이었는데 캐시를 사용하는 제안 방법과 비교하기 위해 그래프에 표기하였다. 제안 방법의 처리량은 캐시리스트 크기 43에서 최대 4.69Mpps이다. 실험 결과를 통하여 제안 방법은 기존 방법에 비하여 최대 초당 790,000개의 패킷을 더 처리할 수 있으며 처리량이 약 20.21% 증가하였음을 확인할 수 있다.

Fig. 13은 제안방법에서 사용하는 메모리 사용량을 캐시리스트 크기별로 측정한 실험결과이다. 캐시를 사용하지 않는 기존 트라이 방법에서 사용한 메모리 사용량은 32,356kbyte이었다. Fig. 14는 캐시를 사용함으로써 증가한 메모리 사용량 대비 성능 증가분을 캐시리스트 크기별로 나타낸 그래프이다. 캐시리스트 크기가 11일 때, 가장 효율적인데 메모리는 기존 방법 대비 2,508kbyte가 증가하였고 성능은 기존 방법 대비 0.705Mpps가 증가하여 kbyte당 처리량 증가 비율은 281.10이다. 이 경우 메모리는 기존 방법 대비 7.74% 증가한 셈이고 처리량은 18.08% 증가한 셈이다.

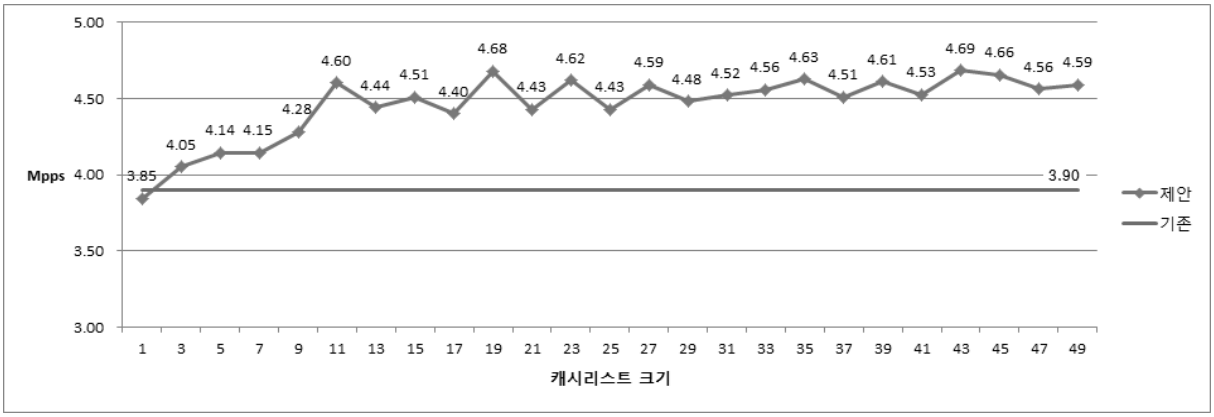

Fig. 12. Experiment Result of Throughput

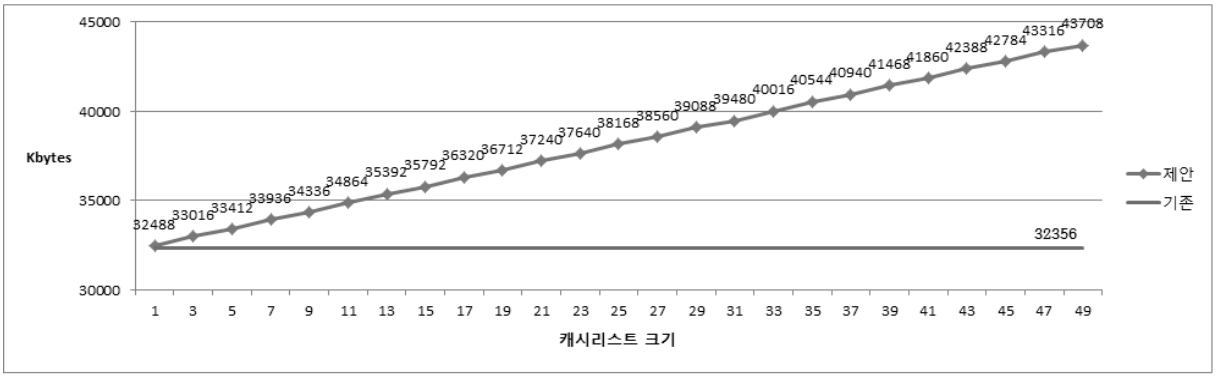

Fig. 13. Experiment Result of Total Memory Size

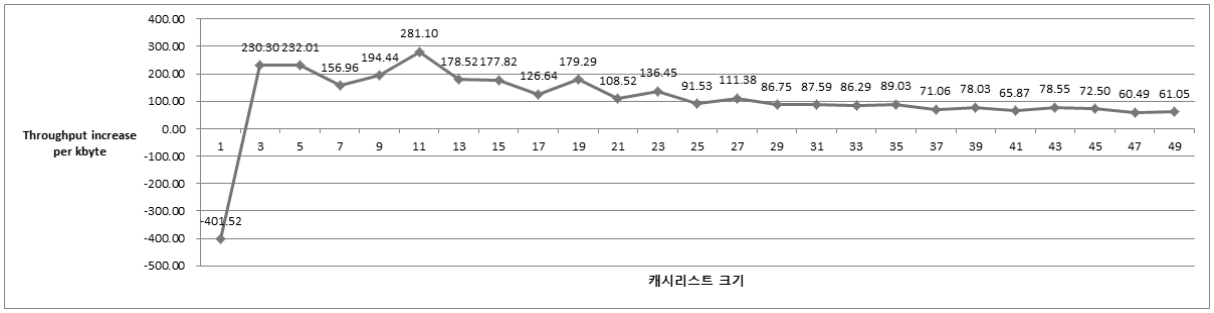

Fig. 14. Throughput Increase with respect to Memory Size Increase

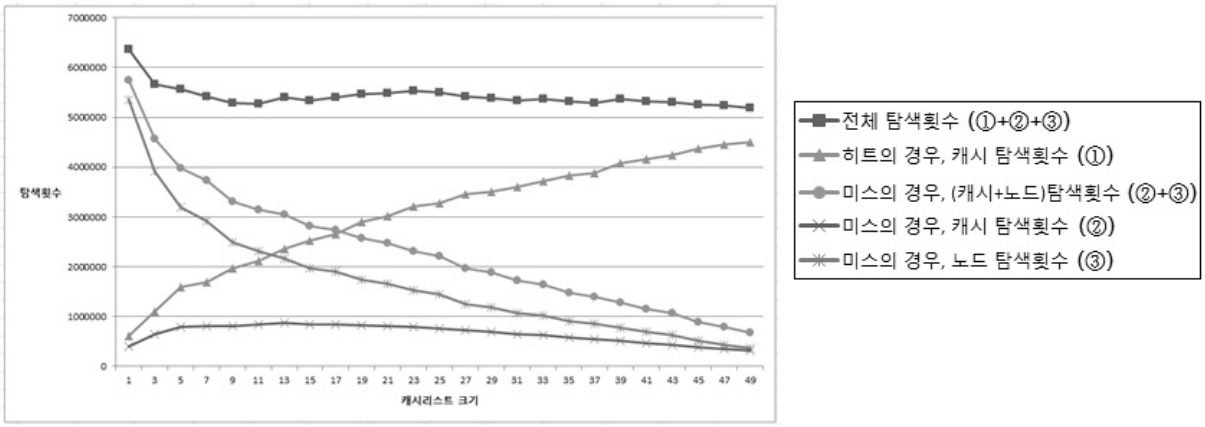

Fig. 15. Total Number of Search in the Experiment

Fig. 12에서 보면, 캐시리스트 크기가 11일 때까지 점진적으로 처리량이 증가하지만, 캐시리스트 크기 13 이상부터는 포화(saturation) 현상이 나타나는 것을 확인할 수 있다. 포화 현상의 원인을 분석하기 위하여 각 실험에서 수행된 탐색횟수들을 조사하였으며, 그 결과는 Fig. 15와 같다. 캐시 히트 되었을 경우의 캐시리스트 탐색횟수, 캐시미스 되었을 경우의 캐시리스트 탐색횟수 및 트라이 노드에서 수행한 이진탐색횟수를 조사했는데 이들 탐색횟수의 총합이 해당 실험에서 전체 탐색횟수에 해당한다.

Fig. 15는 캐시리스트 크기별로 탐색횟수를 나타낸 그래프이다. 그래프에서 캐시리스트 크기가 커질수록 캐시히트의 탐색횟수(범례 ①)가 증가하며, 미스의 경우 탐색횟수(범례 ②+범례 ③)는 감소하는 것을 확인할 수 있다. 또한, 전체 탐색횟수(범례 ①+범례 ②+범례 ③)가 캐시리스트 크기 11까지 점진적으로 감소하는 것을 확인할 수 있으며, 캐시리스트 크기 13 이상부터는 전체 탐색횟수가 포화 현상 패턴을 보이는 것을 확인할 수 있다. 캐시리스트 크기 1부터 11까지 점차적으로 전체탐색횟수가 감소하는 이유는 캐시히트의 탐색횟수(범례 ①) 증가량에 비하여 캐시 미스의 탐색횟수(범례 ②+범례 ③) 감소량이 더 크기 때문이며, 캐시리스트 13 이상의 경우는 캐시히트의 탐색횟수 증가량 대비 캐시미스의 탐색횟수 감소량이 비슷하기 때문에 포화 현상이 발생한다. 즉, 처리량 결과에서 캐시리스트 크기 13 이상부터 포화 현상이 발생하는 이유는 탐색 횟수의 포화로 인한 것이다.

또한, 캐시리스트크기 1인 경우는 기존에 비하여 처리량이 약간 낮은 것을 확인할 수 있다. 이러한 이유는 캐시리스트를 추가함으로써 얻어지는 이득(캐시히트)보다 추가 수행되는 동작에 대한 오버헤드가 더 큰 것으로 보인다.

실험을 통하여, 제안 방법은 기존 방법에 비하여 최대 초당 790,000개의 패킷을 더 처리할 수 있음을 확인할 수 있었다. 제안하는 방법이 기존 방법에 비하여 더 높은 처리량을 나타내는 이유는 다음과 같다. 첫 번째 이유는 클라이언트와 서버 사이에서 한 번에 다수의 패킷 교환이 발생함으로써 패킷이 캐시 리스트에 히트될 수 있었기 때문이다. 두 번째 이유는 캐시리스트 미스가 발생하더라도 캐시 리스트에서 탐색된 값들을 추가 수행될 이진 탐색의 범위 설정에 사용하여 이진 탐색의 범위를 축소시킬 수 있었기 때문이다.

5. 결 론

본 논문에서는 네트워크 필터링의 탐색 방법 중의 하나인 트라이의 성능을 개선하기 위하여 캐시를 적용한 트라이를 제안하였다. 클라이언트와 서버 사이의 패킷 교환에서 한 번에 다수의 패킷이 송수신되는 경우에 대하여, 기존 방법은 동일한 탐색을 반복적으로 수행하는 단점이 있다. 반면, 본 논문에서 제안하는 방법은 기존 방법에 캐시를 적용함으로서 불필요한 반복 탐색을 방지할 수 있다. 제안하는 방법은 캐시를 우선순위로 탐색함으로써 네트워크 필터링 성능이 향상되는 것을 확인할 수 있다.

제안방법의 성능 개선 효과를 검증하기 위해 2만 개의 필터를 적용하는 네트워크 필터링 실험을 수행하였다. 서버 30대와 클라이언트 50대로 구성된 클러스터 환경에서 SPECweb을 수행하여 실험에 사용할 트래픽 패턴을 수집하였다. 테스트 툴로는 Netmap을 사용하였다. Netmap에서 제공하는 packet generator에서 Sender 측을 수집한 트래픽 패턴을 반영할 수 있도록 수정하였고, Receiver 측을 본 논문에서 제안하는 필터링 알고리즘을 적용할 수 있도록 수정하였다. 실험결과에 의하면, 제안 방법은 기존 방법에 비하여 최대 초당 790,000개의 패킷을 더 처리할 수 있었다. 캐시리스트 크기가 11일 때, 캐시 사용으로 인한 메모리 사용 증가량 대비 성능 개선이 가장 우수하였으며, 그때 처리량 증가는 18.08%이었고 메모리 사용량 증가는 7.75%이었다. 메모리 사용량을 고려하지 않고 최대 성능 개선을 보인 경우는 캐시리스트 크기가 43인 경우로서 처리량이 20.21% 향상되었다.

Reference

[1] “Cisco VNI(Visual Networking Index) 2013”, http://www. cisco.com/

[2] Pankaj Gupta, Nick McKeown, “Algorithms for Packet Classification”, Network, IEEE Vol.15, No.2, 2001.

[3] P. Tsuchiya. “A search algorithm for table entries with non-contiguous wildcarding”, unpublished report, Bellcore, 1992.

[4] Chang, Yeim-Kuan, and Hsin-Mao Chen, “Set Pruning Segment Trees for Packet Classification.” Advanced Information Networking and Applications (AINA), 2011 IEEE International Conference on. IEEE, 2011.

[5] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel, “Fast and ScalableLayer four Switching”, Proceedings of ACM Sigcomm, September, 1998.

[6] Srinivasan, Thanukrishnan, et al., “Supervised grid-of-tries: a novel framework for classifier management.”, Distributed Computing and Networking. Springer Berlin Heidelberg, 2006.

[7] iptables, http://www.netfilter.org/

[8] Gregor N. Purdy, “Linux iptables”, O'REILLY.

[9] Jozsef Kadlecsik, György Pásztor, “Netfilter Performance Testing”, http://people.net-filter.org/

[10] Thomas Heinz, “HIPAC High Performance Packet Classification for Netfilter”, University des Saarlandes, 2004.

[11] Michael Bellion, “nf_HIPAC High Performance Packet Classification for Linux Netfilter”, http://www.hipac.org/

[12] “SPECweb2005 User's Guide”, http://www.spec.org/web2005/ docs/

[13] “SPECweb2005 Benchmark Design Document”, http://www.spec.org/ web2005/docs/

[14] “TCPdump Documentation”, http://www.tcpdump.org/

[15] Netmap, http://info.iet.unipi.it/~luigi/netmap/

[16] Luigi Rizzo, “netmap: a novel framework for fast packet”, Proceedings of the 2012 USENIX conference on Annual Technical Conference, 2012.

[17] Luigi Rizzo, Marta Carbone, Gaetano Catall, “Transparent acceleration of software packet forwarding using netmap”, INFOCOM, 2012 Proceedings IEEE, 2012.

김 호 연

e-mail : hykim@q.ssu.ac.kr

2011년~2013년 펌킨 네트웍스/개발 엔지니어

2012년 숭실대학교 정보통신전자공학부 (학사)

2014년 숭실대학교 정보통신공학과 석사

관심분야 : 네트워크 컴퓨팅 및 보안

정 규 식

e-mail : kchung@q.ssu.ac.kr

1979년 서울대학교 전자공학과(공학사)

1981년 한국과학기술원 전산학과 (이학석사)

1986년 미국 University of Southern California(컴퓨터공학석사)

1990년 미국 University of Southern California(컴퓨터공학박사)

1998년 2월~1999년 2월 미국 IBM Almaden 연구소 방문연구원

1990년 9월~현 재 숭실대학교 정보통신전자공학부 교수

관심분야 : 네트워크 컴퓨팅 및 보안